Larch/Aldor—A Larch BISL For AXIOM and Aldor

A thesis submitted to the

UNIVERSITY OF ST ANDREWS

for the degree of

DOCTOR OF PHILOSOPHY

by

Martin N Dunstan

School of Mathematical and Computational Sciences

University of St Andrews

September 1999

Abstract

Computer algebra systems (CAS) such as **axiom** and Maple are programs that have been designed to help humans to solve algebraic problems using symbolic methods. They are often large systems containing libraries developed by different people at different times, and they generally provide an object language to allow other users to extend the system. However, even though the library components may be implemented correctly, there is a risk that they may not be used correctly by the user or other developers. For example, pre-conditions that are not documented or are ignored may lead to inappropriate usage and subsequent failures may have disastrous results.

In this thesis we investigate the use of lightweight formal methods and verification conditions (VCs) to help improve the reliability of components constructed within a computer algebra system. We follow the Larch approach to formal methods and have designed a new behavioural interface specification language (BISL) for use with Aldor: the compiled extension language of **axiom** and a fully-featured programming language in its own right. We describe our idea of lightweight formal methods, present a design for a lightweight verification condition generator and review our implementation of a prototype verification condition generator for Larch/Aldor.

We also describe three case studies that we have undertaken during this research. The first examines the use of VDM reification techniques to derive efficient **axiom** programs for computing the strengths of spectral lines of hydrogen-like atoms. The other two case studies examine the effectiveness of our lightweight verification techniques and identify issues which affect their use.

I, Martin Dunstan, hereby certify that this thesis, which is approximately 60,000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

date (to be filled in) *signature of candidate* (to be filled in)

I was admitted as a research student in October 1995 and as a candidate for the degree of Doctor of Philosophy in October 1996; the higher study for which this is a record was carried out in the University of St Andrews between 1995 and 1999.

date (to be filled in) *signature of candidate* (to be filled in)

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date (to be filled in) *signature of supervisor* (to be filled in)

In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any *bona fide* library or research worker.

date (to be filled in) *signature of candidate* (to be filled in)

Acknowledgements

I would like to thank Ursula Martin for all the encouragement and guidance that she has given to me throughout this research. I must also thank Andrew Adams, James Davenport, Tony Davie, Mike Dewar, Hanne Gottliebsen, Tom Kelsey, Steve Linton, Duncan Shand and Simon Thompson for many interesting discussions. Thanks also to Helen and Joy.

Finally thanks to my wife Vivienne—although she has endured a lot during the past few years, she has always been there for me. I am grateful for her support and her penetrating questions, and I am always amazed by her fortitude and determination never to give in.

Contents

Chapter 1

Introduction

1.1 Aims and motivation

Our motivation for this research stems from the potential unreliability of computer algebra systems (CAS) and the perceived need to utilise formal methods to increase their reliability, and also the reliability of extensions that have been constructed using CAS libraries. We assume that most, if not all, of the components of a CAS perform their tasks correctly otherwise users would be unlikely to continue using such systems. We assume that these components have been implemented correctly and are comforted by the fact that the mathematics which lies behind them is likely to have been studied and reviewed extensively in the literature; some algorithms may even pre-date the electronic computer. However, even if the CAS library functions are all correct, the programs which are built from them might not be. For example, a user or developer might select the wrong function for a task which may cause unexpected and potentially disastrous failures.

Formal methods can help to deal with mistakes in CAS in two ways. Firstly formal methods techniques can be used to provide runtime support for calculations made by the CAS. This may involve interactions between the CAS and an automated theorem prover, the latter using specialised search techniques to attempt to prove or disprove statements that are presented to it. Secondly, formal methods can be used in a software engineering context by providing a way to construct more reliable CAS software by preventing errors from being introduced during the design and implementation phases. These formal methods techniques

may be able to help the user to discover errors at any point in the lifetime of the software and provide a reliable way of ensuring that any fixes do not introduce other mistakes. It is the software engineering approach that we concentrate on in this thesis.

Computer algebra systems are big and complicated, often comprising of a large corpus of source code that was created by different people at different times. As a result we believe that a fully rigorous formal verification of each line of code of a state-of-the-art CAS such as Maple [13] or **axiom** [48] is infeasible.

We propose a lightweight approach where an annotation language is used to define behavioural interface specifications (BISLs) [37] and where the annotations are used to support the analysis of function applications and procedure calls. Such analysis may generate a set of verification conditions (VCs)—statements which, if valid, guarantee that the program satisfies its specification. We leave it to the user to decide what to be done with the VCs—some may be trivial and could be proven correct automatically but others may be too complex for current theorem proving technology. The user may wish to apply specialist knowledge to convince themselves that the verification conditions are valid, or may simply record them as extra conditions on the use of their program. There may be conditions which are obviously false and do not need detailed analysis to show this. For example, although attempting to prove or disprove statements about continuity is undecidable in general, the verification condition

$$\tan(x)\ isContinuousOn\ (0, \pi)$$

is clearly false and can be seen by drawing a graph of $\tan(x)$ over the range $(0, \pi)$.

Our aim is not to provide a fully verified system but a methodology and tools by which more reliable systems can be constructed. The techniques that we describe in this thesis may be utilised not just by the CAS designers but also by any developer of computer algebra routines. They may also be useful in the wider context of software engineering and are not restricted to the specific programming and specification languages that we have used. Indeed other programming languages may benefit more from our approach than ours is able to, while other specification languages may provide better features.

In this thesis we focus on the use of the Larch approach to formal methods [37], and apply them to the **axiom** CAS [48] and its compiled library language Aldor [83].

1.2 Results and achievements

In the main part of this thesis we describe in detail the results of our research but we also provide a brief summary here. Our work began with a case study into the application of VDM reification techniques [50] to programs written for computer algebra systems. The particular study centred around the computation of the strengths of spectral lines of hydrogen-like atoms. Our aims and objectives were to:

- develop an **axiom** program from a VDM specification
- use reification to select appropriate data-types for representing real numbers
- repeatedly reify the program to obtain more efficient (faster) implementations
- investigate various correctness issues

We were successful in achieving these objectives and describe our methods and results in more detail in Chapter 3. We also comment on the use of specification matching [85, 88] as a potentially useful technique for locating functions by their rather than their name.

The core of our research is divided into two parts. The first part is the design of a new Larch [37] annotation language for Aldor [83] which is described in Chapter 4. Aldor is a programming language designed for the efficient implementation of computer algebra algorithms and used in the development of **axiom** libraries. It was previously known as AXIOM-XL and A$^\sharp$, and we provide a brief introduction to the language in Section 1.3.6 and Appendix A. The second part of our research is the development of lightweight formal methods and the *rôle* of Larch/Aldor within it: this is described in Chapter 5.

The syntax of the new annotation language, Larch/Aldor, is based on that of existing Larch annotation languages with new ideas for the annotation of any program statement instead of just functions and procedures. In addition a store model for Aldor was written in an algebraic specification language following work of Chalin on LCL [12].

Two case studies were made to examine the effectiveness of our lightweight verification techniques and to identify issues which affect their use. Although most of the verification conditions generated during these studies were relatively simple, it became apparent that the analysis of polymorphic functions may generate verification conditions that can only be investigated properly when the value of all type parameters are known. We are unsure of the best way to deal with this problem but an approach of keeping “unresolved” verifi-

cation conditions around until the values of unknown types have been identified seems to be feasible. Thus we have the concept of "pending" verification conditions.

On the practical side, we have implemented a simple global data-flow analyser for Aldor programs and a verification condition generator for Larch/Aldor programs. Both tools are written in Aldor and consist of 3600 and 8500 lines of code respectively. Without access to the source of the Aldor compiler (kindly provided by NAG Ltd) it would have been almost impossible to have written the verification condition generator in the time available.

The data-flow analyser uses a three-valued logic to represent whether program variables have, have not or might have, been defined/declared/used. This analysis provides the user with more information than the current version of the Aldor compiler and may be useful for detecting a certain class of subtle programming mistakes. The implementation is naïve since it does not make use of standard techniques such as representing programs as graphs and recording the states of variables with bit vectors. Instead programs are analysed from recursively building a hash table of information about each identifier in a given scope level. It is well known that global program analysis such as this is inefficient!

However, the knowledge gained from the implementation of the data-flow analyser was an important step towards the design of the prototype lightweight verification condition generator described in Chapter 5. Before we were able to create this tool we needed to make a few modifications to the Aldor compiler so that the syntax analyser could recognise Larch/Aldor annotations. It was also necessary to extend the compiler so that it could generate a textual representation of an Aldor program such that every symbol was attributed with its type and any Larch/Aldor annotations. The benefit of obtaining a textual representation of annotated programs was that the verification condition generator did not have to perform any syntax or type checking of Aldor programs—this is very important when dealing with a language with a type-system as powerful as that of Aldor.

Our verification condition generator is novel because it uses forward analysis of programs rather than the more common backwards analysis. Although the verification conditions generated by this approach may become large, they do contain as much information as possible about the program being analysed from its source and annotations. Verification conditions are stored internally using conjunctive normal form so that they can be presented to the user as a set of small logical expressions (which are easier to manipulate than a single large term). The current implementation uses the object language of the Larch

Prover [37] to communicate verification conditions to the user, and makes use of comments to allow verification conditions to be associated with a specific line of source code. The modular design of the tool means that the object languages of other theorem provers could be supported relatively easily.

During this project we have learnt that reification can not only be used to help developers to construct implementations from abstract specifications but, in the context of a computer algebra system such as **axiom**, may also be used to develop more efficient solutions. We have also learnt that a lightweight verification condition generator is not unduly difficult to implement—the hardest part was the pre-processing phase that removed the time dependence from the annotations of user programs. Writing down a formal model of the Larch/Aldor store was also non-trivial and writing specifications for frequently used Aldor data-types can be time consuming. However, we believe that lightweight verification condition generation has a lot to offer the user—it enables them to concentrate on the formal development of parts of the project which are likely to benefit the most. Coupled with the specialist knowledge of the user, our techniques may help them to discover errors in programs without resorting to a completely formal development in which the validity of every verification condition is checked.

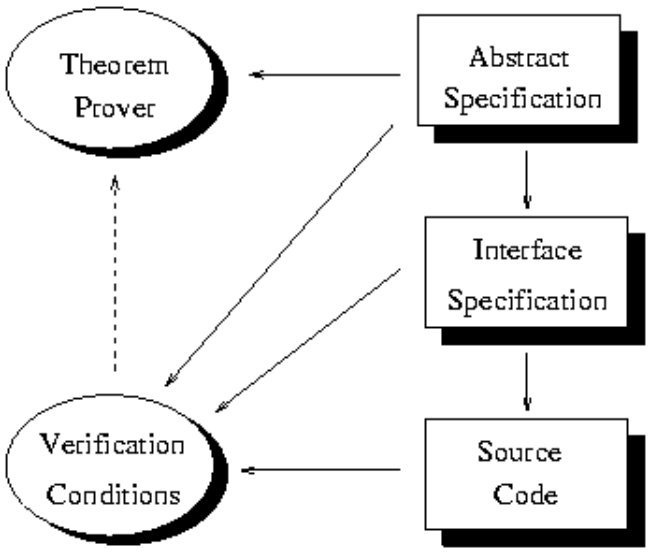
Related to our work is that of Kelsey [52] who has modeled the **axiom** category hierarchy of mathematical structures and a number of interesting **axiom** functors using LSL, the algebraic specification language of Larch [37]. These specifications can be used to provide the background theory for Larch/Aldor programs by defining the semantics of operations and data-types which are available to the Aldor programmer. In addition, the LSL theories can be used with proof attempts of any lightweight verification conditions generated by our tools. Specification libraries of this kind are essential if our techniques are to be applied to CAS such as **axiom**.

1.3 Context of this research

Our research focuses on the use of formal methods (in particular specifications) to increase the reliability of computer algebra systems and thus increase the confidence of their users that the answers they receive are the “correct” ones. In this thesis we concentrate on the use of the Larch approach to formal specification and software development and apply it

to Aldor, the library programming language for the **axiom** computer algebra system. The result of this is the Larch/Aldor BISL which is described in more detail in Chapter 4 and a prototype verification condition generator (see Chapter 5).

The diagram below illustrates the development process for programs constructed using our approach. First abstract specifications are written to provide the background theory for the problem being tackled and may be investigated using appropriate tools such as a theorem prover. Next the program source code is written and annotated using interface specifications to link the abstract specifications to the chosen implementation. Finally a verification condition generator may be used to help detect mistakes in the implementation.



In Section 1.3.1 we provide a brief introduction to program specification and Larch. Then in Section 1.3.4 we introduce computer algebra systems (CAS) in general and **axiom** in particular. Finally in Section 1.3.6 we talk about the Aldor programming language—a more detailed description of which can be found in Appendix A.

1.3.1 Program specification

It is generally accepted that specifications are a “good thing”: without a specification of the problem it may not be obvious what is to be solved and in which direction one ought to proceed. Indeed, how can one decide whether a problem has been solved correctly if one

does not have a specification of what it is that we are trying to achieve or how the resulting system is to expected to behave.

Specifications may be written in a variety of languages ranging informal descriptions in plain English, through to mathematical notations such as VDM [50] or Z [72]. Each language has its benefits and drawbacks. For example, prose is useful for conveying a meaning to a wide range of readers who wish to know the overall picture but it is often likely to omit significant pieces of information, be too imprecise or contain contradictions: see, for example, the problems with the definition of the Algol programming language identified by Knuth [55]. At the other end of the spectrum, formal specification languages are designed to be precise and amenable to mathematical analysis: this allows specifications to be checked for omissions and inconsistencies but may mean that the casual reader is unable to understand their meaning.

1.3.2 Using specifications

Once a specification has been written it can be used to guide an implementation. To achieve the transition from specification to program code various techniques have been utilised to ensure that the implementation is correct. Early techniques were based around the work of Floyd [28] and Hoare [42] where a program is decomposed into a set of mathematical formalæ called verification conditions using axioms and rules of inference. If all the verification conditions can be proved to be true then the implementation is considered to be correct. However, in an international survey of industrial applications of formal methods made by the US Department of Commerce [16] points out that "*complex language features result in complex proof rules*" and that "*proof rules are exceedingly difficult to get right*".

Applying the Floyd-Hoare technique in its entirety to small programs is not easy and we argue that it is not suitable for large programs. Indeed once a mistake is discovered in the implementation it may be too late to remove it with ease. One example of a system for the development and verification of programs using a variant of Floyd-Hoare Logic is the Gypsy Verification Environment [16, pages 88–89]. Amongst other things this system includes a tool for the generation of verification conditions from programs written using the Gypsy specification and programming languages. A mechanical proof-checker can be used to help with proof attempts. The philosophy of the Gypsy approach is that the development of the program, its specification and correctness proofs ought to be performed together as

an iterative process. This is contrasted with the approach of specifying, implementing and then verifying the result.

More recently attention has been directed to proving properties of specifications to determine whether or not they address important aspects of the problem domain. Once the specification is deemed to be ready an automatic translation into an implementation is made such that the resulting program is guaranteed to satisfy the specification. RAISE [16, pages 101–102] is one system that uses this approach—specifications may be transformed in various ways such as changing types and removing under-specification. Each transformation may generate intermediate proof obligations that must be discharged and in this respect it is similar to the VDM reification methodology used in Chapter 3. The main differences are that RAISE provides tools for automatically generating these proof obligations and has tools which partially mechanise the generation of C or Ada programs from specifications. Other systems such as the COQ theorem prover [3] allow programs to be generated automatically from proofs. This follows from the "proofs-as-programs" equivalence available to users of constructive logics. However, these systems may place excessive demands on the developer, especially for large systems, and the resulting programs may not be very efficient. They do not address the issue of legacy code either.

Finally there is the lightweight approach described in this thesis. We take a more pragmatic approach whereby the developer is encouraged to write good specifications and perhaps prove properties about them. The reification approach of Chapter 3 may be used to assist the translation of specifications into source code or perhaps other techniques may be applicable instead. Annotations in the source code permit tools to generate verification conditions which help to ensure that library functions are used in the correct manner. If a top-down development process is adopted then lower-level functions can be defined with an annotation describing their intended behaviour but no implementation. The verification conditions can be used to investigate properties of the annotated program and changes to the specification of the behaviour of the low-level building blocks may be altered before they are actually implemented. Such techniques allow mistakes to be detected as early as possible and thus reduce the cost of fixing them.

The lightweight approach can also be used with legacy code by following the techniques described by Evans in [25]. The existing code can be annotated with specifications of its perceived behaviour and then verification conditions generated from them can be used to investigate whether the annotations are detailed enough or whether they contain incon-

sistencies. This process can be repeated until a significant part of the program has been annotated and its behaviour understood to a sufficient degree.

1.3.3 Larch

One particular issue which relates to formal specification languages and their use with programming languages is that of generality. That is, does the specification language allow programs in a wide variety of programming languages to be described or is it specific to just a few? This is important because a wide-spectrum specification language such as Z [72] can be used to model important properties of the problem but the resulting specifications may be too abstract. This may mean that it is difficult to produce an implementation and check that it satisfies the specification. In contrast to this, a specification language that is strongly tied to a particular programming language may not provide enough abstraction and therefore the benefits of formal specification may be lost.

The Larch approach [37], primarily developed by John Guttag *et al.* at MIT, tackles this problem by adopting a two-tiered system. The first tier is represented by a programming-language independent algebraic specification language called the Larch Shared Language (LSL) while the second tier consists of a family of Behavioural Interface Specification Languages (BISLs), each tailored to a particular programming language.

LSL—The Larch Shared Language

LSL specifications are intended to provide the background theory for the problem domain and the semantics for symbols and types which appear in BISL specifications. A tool is available to perform syntax and type checking of LSL specifications and to convert them into the object language of the Larch Prover (LP), a proof assistant for first order logic. Although LP has been used to investigate and prove theorems in a variety of problem domains such as the verification of the correctness of a digital circuit [15], reasoning about algebraic theories [64] and verification of provably correct compilers [76], it is often used to check that LSL specifications have the properties that their creator expects them to.

LSL specifications are composed from traits which are divided into several sections. In the diagram below we give an example of a trait for the natural numbers: in the `introduces`

```
Natural: trait
   introduces
      0, 1    : → N
      succ    : N → N
      __ + __ : N, N → N
   asserts
      N generated by 0, succ
      ∀ n, m:N
         ¬(succ(n) = 0);

         1           == succ(0);
         0 + n       == n;
         n + 0       == n;
         n + succ(m) == succ(n + m);
   implies ∀ n:N
      n + 1 = succ(n);
```

section of this trait new symbols 0, 1, the successor function `succ` and binary addition `+` are introduced with their signatures. The symbol `N` appearing in the signatures represents an LSL sort (a type) and is used here to represent the naturals. The `asserts` section asserts various properties—the `generated by` line states that all values of the sort `N` can be constructed using applications of `0` and `succ`; it also allows proof-by-induction over `N`. The remainder of the `asserts` section defines a set of axioms designed to capture the properties of the natural numbers. The final `implies` section allows the specifier to define other properties which follow from the axioms in the `asserts` section. These can be regarded as lemmas and must be proved before they can be used in other traits.

Other features of LSL include the ability to parameterise traits by symbol and sort names, include or assume properties from other traits, define equality over sort values and to indicate whether a predicate has been completely specified. The latter property is necessary because LSL symbols may be under-specified, either by accident or deliberately to specify partial functions such as division.

BISLs—Behaviour Interface Specification Languages

Larch BISLs are a family of annotation languages, each tailored to a particular programming language and are based around the concept of pre- and post-conditions. User programs are annotated in the appropriate BISL and are able to make use of operators defined

in LSL theories. BISL specifications are primarily concerned with implementation details such as side-conditions on functions, memory allocation and pointer dereferencing. In the ideal world users would annotate their programs at the same time (or just before) they construct the implementations, just as one ought to do with comments. However, this may not always be possible, especially when working with legacy code.

At the time of writing there are about 11 different Larch BISLS for languages ranging from CLU [84] and Modula-3 [51] to C [37] and C++ [59]. Each has been designed to investigate particular aspects of imperative programming such as inheritance and concurrency as well as different development methodologies such as specification browsing [14] and interactive program verification [35]. The syntax and use of BISL specifications is essentially the same for all languages—functions and procedures may be annotated with statements defining their pre- and post-conditions as well as indicating any client-visible objects which *might* be modified when the function is executed. A review of different Larch BISLs is given in Chapter 4.

1.3.4 Computer algebra systems

Computer algebra systems are environments for symbolic calculation which have been designed to help humans to solve various kinds of algebraic problems symbolically. They allow users to manipulate of expressions involving symbols which might, at some point, be assigned concrete numeric values. General purpose computer algebra systems such as Maple [13], **axiom** [48] and Mathematica [86] also provide facilities for graphical display of curves and surfaces, and most may be extended via a system-specific programming language. There are also more specialised tools such as GAP [31] for computational discrete mathematics and libraries such as **axiom**/PoSSo for high-performance polynomial system solving. These systems are used by many different communities of users including school and university teachers, engineers, and researchers in both science and mathematics. Aerospatiale, for example, used a Maple-based system for motion planning in satellite control [73] while Brown [9] used **axiom** for elaborate computations in group theory. The specialised systems in particular are extremely powerful: the PoSSo library has been used to compute a single Gröbner basis which occupies more than 5Gb when compressed, while GAP is often used to compute with groups of permutations on millions of points.

A classification and review of computer algebra systems can be found in [81] while [10] is

a survey of applications for computer algebra by the same authors.

1.3.5 axiom

axiom [48] is a general purpose, strongly-typed computer algebra system which began life at IBM in the mid-70s as a system called Scratchpad and is now maintained by NAG in Oxford, England. The versions used during this research are 2.0 and 2.1. The system can be used interactively via a textual interface and graphical results may be displayed in a separate window. All values entered by the user and returned by **axiom** have a unique type and thus enables users to quickly spot mistakes resulting from unclear or incorrect input. In addition the type checker can prevent operations being applied to inappropriate values. For example, when `2*x**3 - 4*x + 1` is typed into **axiom** it will display the result $2x^3-4x+1$ and indicate that its type is `Polynomial Integer`. If the user really wanted a polynomial with floating-point coefficients then they can retype the expression and insist that the result has type `Polynomial Float`. Functions and types are first class values and may be used just like any other value provided that type-correctness restrictions are satisfied. Programs may be written in the interpreted **axiom** interactive language or in Aldor, a compiled extension language (see Section 1.3.6).

The most notable feature of **axiom** is the two-level object model of *categories* and *domains*. Categories are conceptually equivalent to the mathematical or logical notion of a category and are used to specify information about domains. A category declares the names and types of values (usually functions) that a domain will provide to the user. They are closely related to Haskell type classes [44] and may be parameterised by arbitrary values; they can be joined to form new categories and can inherit from one or more other categories. The intended meaning of a category is defined by its name, by the symbols it declares and by documentation comments in the source code. For example, the intended meaning of the `SemiGroup` category is apparent from its name. From its documentation we find that it is *"the class of all multiplicative semigroups i.e. a set with an associative operation* $*$*"*. Definitions in a category may be conditional on the properties of any parameters; default values may also be provided. It is interesting to note that Java [27] has a similar concept known as an `interface`. However, Java interfaces cannot be parameterised by values, nor can methods be defined conditionally. Like Java, categories with no body can represent typed attributes for domains.

A domain is a type which defines zero or more exported (publicly accessible) symbols. These symbols correspond to constant values and are usually functions although they do not have to be. A domain is an instance of a single category and may be parameterised by arbitrary values; domain exports may be conditional if the category being satisfied requires this. Thus domains are similar to Haskell `instances` and Java or C++ (concrete) classes. If a domain defines a representation then we call it an *abstract data type* otherwise it is called a *package*. Domains may be tested to discover which categories they belong to and this is often used when defining conditional exports. For example, domains which define an ordering on their elements may claim to belong to the category `Order`; domains representing collections of values of type `T` may test if `T` belongs to `Order` and provide a sorting operation if it does.

It is important to note that **axiom** cannot prove, for example, that a domain which claims to belong to the `SemiGroup` category really is the same as a mathematical semi-group. However, if the implementers of **axiom** domains ensure that such correspondences are retained, then the strong type system will ensure that the user does not attempt to perform inappropriate operations. For example, the `Polynomial(R)` domain constructor requires that its argument `R` belongs to the category `Ring` and **axiom** will allow the construction of `Polynomial(Integer)` since `Integer` satisfies `Ring`, but it will disallow the construction of `Polynomial(Boolean)` since `Boolean` does not.

The reader is referred to Davenport [18, 19] for an excellent description of Scratchpad and the construction of abstract algebra in such a system; by the same author [17] describes **axiom** as it is now.

1.3.6 Aldor

Aldor [83] was designed to be an extension language for **axiom** in which computer algebra routines could be naturally and efficiently implemented. It is a type-complete, strongly-typed, imperative programming language that has a two-level object model just like that of **axiom**. Types and functions are first class entities allowing them to be constructed and manipulated just like any other values; dependent types can be used in certain contexts such as function definitions to provide parametric polymorphism. For example, the function to sum a list of elements might be declared to be of type `(R:Ring, l:List R) -> R`. The first argument `R` is a type which satisfies the category `Ring` and so there must be an

addition operator `+` of type `(R, R) -> R` and a zero element `0` of type `R`. The type of the second argument and the type of the return value of this function both depend on the value of the first argument `R`; static type checking ensures that this function can only be applied if the first argument is a type satisfying `Ring`.

A novel feature of Aldor are *post facto extensions* which allow the meaning of existing types to be extended. This mechanism is used by existing Aldor libraries to create domains in several stages starting with basic implementations which are later extended with more operations to increase their functionality. Other features include automatic garbage collection, generic iterators and interoperability with languages such as C, C++, Fortran and LISP. It is possible for example, for an Aldor function which invokes a C function to be passed as an argument to a Fortran procedure. This interoperability enables Aldor programs to be written that use existing libraries such as the NAG Fortran library or the C++ PoSSo library, and for programs written in these languages to utilise Aldor libraries.

An example of a simple Aldor category to model sets might be written:

```
define SetCategory(T:BasicType): Category == with
{
   -- We inherit the operations of finite linear aggregates
   FiniteLinearAggregate(T);

   -- Declarations: % is the domain which implements us
   member?  : (T, %) -> Boolean;
   subset?  : (%, %) -> Boolean;
   union    : (%, %) -> %;

   -- Exports such as intersection() omitted for brevity
}
```

Using this category we can provide an implementation for the domain of sets:

```
Set(T:BasicType): SetCategory(T) == add
{
   Rep == HashTable(HashKey, T); -- Internal representation
   import from Rep; -- Make the hash table operations visible

   member?(t:T, S:%): Boolean ==
   {
      -- Search the internal rep using the key hash(t)
      (found, value) := search(rep(S), hash(t), t);
      found;
   }

   -- Other exports omitted for brevity
}
```

A more detailed introduction to Aldor can be found in Appendix A, but Watt [83] is the definitive guide to the language. A strongly-typed embeddable computer algebra library called $\sum^{\text{it}}$ has been implemented in Aldor and is described in [7]; a library called $\prod^{\text{it}}$ is currently being designed to allow parallel programs to be written in Aldor [63] where the underlying architecture is abstracted away using categories. Poll and Thompson [71] have produced a formal description of the Aldor type system.

1.4 Related work

1.4.1 Program specification and program checking

As mentioned in Section 1.3.3 there are a number of different Larch BISLs in existence, each designed to allow the specification different aspects of computing to be investigated. For example, Larch/SmallTalk [14] has been used to examine specification browsing while Larch/C++ [59] concentrates on the issues of inheritance of code and inheritance of specifications. However, many of the Larch BISLs do not have any program analysis tools associated with them—they are primarily used as a means to support clear and concise documentation. The exceptions are Larch/Ada [35] and Larch/C [24]. Larch/Ada uses a syntax-directed editor called Penelope [35] for the interactive development and verification of Larch/Ada programs (see Section 4.1.1). Larch/C has a static checker called LcLint [24] which can check for violations of the **modifies** clause but does not attempt use the annotations to produce VCs (see Section 4.1.1). Speckle [82] is also of particular interest (see Section 4.1.1). It is a compiler which was designed to investigate how specifications could make programs run faster. Speckle accepts CLU programs containing different implementations of procedures which are annotated with Larch-style interface specifications and constructs a logical model of the program using control-flow graphs.

Moving away from the Larch world, the Extended Static Checking (ESC) system [22] was developed at DEC SRC to provide automatic machine checking of Modula-3 programs and the techniques developed for this project are now being applied to Java. ESC is designed to detect violations of array bounds and `NIL` pointer dereferencing as well as deadlocks and race conditions in concurrent programs through the use of simple yet powerful annotations. A special purpose theorem prover has been developed specifically for use with ESC to

enable the tool to run quickly and with as little user interaction as possible.

ProofPower is a commercial tool developed by the High Assurance Team (HAT) at ICL [53] based on the HOL [34] theorem prover and the Z [72] notation for a subset of Ada. Functional requirements of programs are often presented to the HAT as documents containing Z specifications. These can be checked and further developed through the use of ProofPower. Programs can then be prototyped in Compliance Notation using Knuth's "Web" system for literate programming, and refined into Ada programs. Verification conditions from the Compliance Notation are generated as Z specifications using an appropriate tool and can be discharged via formal or informal arguments as required (*e.g.* with ProofPower).

Eiffel [65] is a programming language in which specifications containing pre- and post-conditions are an integral part of the language syntax as are loop invariants and measures. The compiler is able to convert the annotations into runtime checks and the programmer can produce their own exception handlers to deal with situations where the specifications are not satisfied. The Eiffel compiler does not perform any theorem proving and it is up to the programmer to ensure that their implementations satisfy their specifications. However, the ability to detect when assertions are violated at runtime via exceptions is better than the facilities provided by many other programming languages.

Extended ML [75] is another programming language which incorporates specifications in its syntax and semantics. Users can write algebraic specifications describing the properties of functions and use stepwise refinement (*c.f.* reification [50]) to obtain suitable implementations. Advantages of this are that the implementer must repeatedly justify their choices of data-type and algorithm which can produce better solution. Unfortunately this means that everything has to be proved formally at each stage. It is interesting to note that while EML annotations are similar to LSL, the former is able to define the behaviour of EML functions directly whereas Larch BISL annotations define the semantics of function using LSL.

1.4.2 Computer algebra and formal methods

Although we have identified two main ways in which formal methods can be used to improve the reliability of computer algebra systems there has also been some work investigating how computer algebra systems can be used to help with theorem proving. Three main techniques have been investigated: the development of CAS inside a theorem prover

(TP), the extension of an existing CAS with TP support and the cooperation of existing TP and CAS. The first technique was adopted by de Bruijn [20] with the development of AUTOMATH for analysis; more recent work includes the development of constructive reals in LEGO by Jones [49] and CAS algorithms in NuPrl by Jackson [46]. The second technique has not been explored so well although the work of Clarke *et al* [5] with Mathematica [86] and reasoning about power series is impressive. Finally CAS may be used as oracles for TP with varying degrees of trust. Examples include using Maple [13] as an oracle to HOL [34] for the reals [40], and the Isabelle/Maple interface [43]. Adams [1] uses PVS [68] to construct and query a table of symbolic definite integrals enabling them to obtain correct answers where existing CAS do not.

1.5 Thesis structure

This thesis is divided into a further five chapters. In Chapter 2 we motivate our work by describing various problems which affect large software systems. We look at ways in which different types of errors can be prevented from appearing in programs in the first place as well as ways to identify other types of errors which may be present in an implementation.

Chapter 3 is a case study that investigates the use of VDM reification techniques [50] to assist in the construction programs to solve computer algebra problems. Reification is used not only to select appropriate data-types to represent polynomials and the real numbers, but also to deliver successively more efficient (faster) implementations. In the light of our experience from this case study we comment on the use of specification matching as proposed by Wing [85] and Zaremski [88].

The next two chapters discuss the core of our work. Chapter 4 describes the design of the Larch/Aldor annotation language beginning with with a survey of existing Larch languages and discuss the issues involved in the design of a new Larch language. Then we present the syntax of Larch/Aldor followed by a model of its store based on the work of Chalin with Larch/C [12]. In Chapter 5 we detail our lightweight approach to program verification, discuss how lightweight verification conditions generated by this process can be used. We discuss the design issues for tool support and then describe the prototype lightweight verification condition generator that we have implemented. We review the lessons that we have learned from the implementation and suggest how it could be developed further.

Finally in Chapter 6 we investigate the uses of Larch/Aldor and lightweight program verification through two case studies. The first case study follows the development of an annotated Larch/Aldor function implementing the quicksort algorithm from algebraic specifications of sorting linear containers such as lists. Verification conditions are generated from the implementation and each is examined in turn. The second case study begins with an existing Aldor library function for converting textual representations of numbers into values belonging to a particular type such as the integers. The program is annotated with Larch/Aldor specifications and the resulting verification conditions generated from it are investigated. We find that even simple verifications generated from both programs may defeat attempts to mechanically discharge them without guidance. We also discover that verification conditions generated from polymorphic functions present extra difficulties since their proof attempts may need to be postponed until the value of all type parameters are known.

Chapter 2

Motivation

In this chapter we motivate our work: the design of a Larch BISL [37] for Aldor [83] and to construct program analysis tools such as lightweight verification condition (VC) generators. We begin in Section 2.1 by outlining various problems which afflict large software systems and comment on domain-specific problems for CAS such as **axiom** [48]. Then in Sections 2.2 and 2.3 we consider ways in which programming language-specific problems can be addressed. Finally in Section 2.4 we link together the ideas discussed in this chapter and show how they relate to the rest of this thesis.

2.1 Problems with large software systems

The issues of creating, maintaining and extending large software systems are an important aspect of software maintenance and development in industry [8]. They include

- legacy systems—programs which may have been written many years ago and whose documentation, source code and perhaps even the original hardware platform may no longer be available. This obviously creates difficulties for maintainers.
- libraries—a large computer system will undoubtably have a significant number of subroutines which may, or may not, be documented. Indeed the names of procedures that are used may not convey any meaning to the reader without documentation.

- modularity—modular programming such as the object-oriented approach to software development has significant benefits which can help to isolate mistakes and reduce the possible number of changes that are required to fix them. If a programming language does not provide support for this then it is up to the programmers to enforce it as part of their work practices; this may not be sufficient in practice.

- communication—large software systems usually require more than one person to work on them. It is likely that a team of people will design the product, another team will implement a solution and yet another team will test it to check that it is viable. The members of each team evolve with time and individuals may not see the system from inception to deployment and so good communication of information between individuals (*e.g.* through documentation) is essential. After deployment other people may be involved with its maintenance and they need good communication with the designers and implementers (again probably through documentation).

Since CAS are often large systems which provide a wide range of functions it is inevitable that they will suffer from the same kind of problems as those mentioned above. For example, the **axiom** [48] CAS which is still being developed and maintained at the time of writing, can trace its origins traced back to the mid 1970's when it was known as Scratchpad. **axiom** has a large library of computer algebra functions and data types which introduces problems for both the user and the maintainer. Both parties need to be aware of the existing resources that are available to them in the library before developing new ones. However, locating functions of interest can be difficult since their names may not familiar or obvious to everyone. For example, in Chapter 3 we would like to use **axiom** to evaluate the integral

$$\int_0^\infty r^n e^{-\alpha r} dr$$

The obvious choice of function to achieve this might appear to be `integrate` but in fact one needs to use `laplace` instead. To help resolve this kind of problem Wing [85] proposes using interface specifications and theorem proving to identify functions according to their behaviour rather than just their name.

It is also important that users and maintainers are aware of the semantics and potential side-effects of functions that they use—in this thesis we argue that in the context of computer algebra systems with libraries containing a large number of functions that are firmly based in mathematics and algebra, it is more likely that errors will occur through misuse of these

functions rather than in their implementation. We believe that interface specifications can be used to alleviate this problem by enabling the designer and implemented to clearly define the meaning of individual functions, to state explicitly the conditions under which they may be used, and to highlight any side-effects that they might produce.

Many CAS also provide an object language and developers using them will probably encounter other standard problems such as failure to initialise variables or invoking functions with invalid arguments. In the sections that follow we look at various ways in which these problems may be tackled, both through prevention and detection. There are, of course other problems which are specific to CAS such as the issue of what kind of transformations can be applied to expressions typed-in by the user (normalisation) and the distinction between symbols which are place-holders for future values and symbols which represent indeterminates. These are areas in which CAS are prone to producing erroneous or unexpected results but ones which we are not concerned with here. That is not to say that the techniques described in this thesis could not be applied to these problems. For example, the simplification of expressions can be regarded as the application of a transformation function, the behaviour of which ought to be amenable to specification.

2.2 Error prevention

In this section we look at the prevention of errors in relation to programming languages in general. First in Section 2.2.1 we consider how the design of a language can affect the quantity and type of mistakes that can be introduced into its programs. Then in Sections 2.2.2–2.2.5 we investigate how software engineering practices can be used to try and prevent errors from appearing in implementations from the outset. We believe that the first defence against mistakes and flaws in programs is to use appropriate software engineering techniques to prevent errors from appearing in the first place. There are various ways of achieving this but in these sections we just consider four that rely on program specification.

2.2.1 Error prevention by language design

One of the best places for reducing the number of bugs in computer programs lies in the design of the programming language itself. A language designer must perform a delicate

balancing act between providing a complex language with a large degree of functionality and one which is simple to understand but which might lack features that programmers want. However, careful choice of syntax and semantics of the language can prevent certain types of errors occurring, make them harder to create and/or make them easier to detect. For example, in S-algol [66] all variables must be initialised to a legal value when they are declared. While this does not prevent the programmer from using a poor initial value it does encourage them to think carefully about it. Later in Section 2.3.3 we see that the powerful type systems and two-level object model of **axiom** and Aldor allow the programmer to concentrate more on the design and structure of an implementation. This may also reduce the amount of time that needs to be devoted to low-level "hacking" and data-mangling compared to a language such as C or C++ and we argue that the resulting software will be more robust and reliable.

Ghezzi and Jazayeri [32] highlight several properties of programming languages which we have summarised below. These properties ought not only to be considered during the design of a language but also during the selection of a language for a particular implementation. In large projects an inappropriate choice could be expensive.

Writeable

The language ought to be easy to express algorithms in so that the programmer can concentrate on problem solving rather than the implementation. Assembly languages are often cited as being non-writeable since the programmer often has to pay particular attention to register allocation at the expense of overall task being tackled.

Readable

Programs ought to be easy to read and comprehend, both in terms of syntax and semantics. This allows the programmer to detect errors in the code as easily as possible and means that the maintainer has an easier task once the product is finished.

Factoring

If a specific fact is dealt with in a single place in the program then any changes which need to be made to it are highly localised and hopefully well understood. The alternative is that the fact needs to be changed in many different parts of the program which admits the possibility that some parts are missed or some changes are different to others. Most modern languages support this concept through the use of procedures and named constants; some languages also support this by providing abstract data-types.

Abstraction

Data abstraction means that the client of a module or data-type only needs to be aware of its interface, not the internal implementation or representation. This separation of concerns allows the creator to change the module provided that the behaviour and interface remain constant. Since the client can rely on this fact, their task is made simpler by the abstraction and should not have to alter their code to take account of any changes. Control abstractions are similar in concept—they provide a way for the programmer to describe the order in which a collection of statements are executed. They alleviate the problem of determining execution paths through the analysis of jump or branch instructions that plague assembly languages and early versions of Fortran.

Exceptions

Catering for unexpected and undesirable events at the point where they occur is complex and generally produces unreadable and error-prone programs. Furthermore, some events may be so unexpected that the programmer fails to catch them! Exception handling facilities are designed to separate the task of dealing with errors from the main problem solving which often leads to clearer and simpler programs. However, recovering after exceptions may be non-trivial due to the distance between the code that raised the exception and the code which deals with it.

Optimisable

If the programming language is easily optimisable by the compiler or translator then the programmer will not have to spent too much effort in this area. The result should be that programs are easier to read and will contain fewer mistakes since hand-optimisation generally makes programs more opaque and error-prone. A classic example of this is the practice of Fortran programmers tweaking the assembly code produced by early compilers to achieve suitable optimisations!

Other properties

Other language design issues for reducing the number of possible errors in programs include the provision of type information (see Section 2.3.3), redundant keywords to improve syntax checking (see Section 2.3.2) and variable declaration (see Section 2.3.4). One important decision which will not be considered further is the choice between different programming methodologies such as imperative *versus* functional *etc.*

2.2.2 Program specification

As we described in the introduction, specifications are widely recognised as a valuable part of the software engineering process. They are essential for defining the problem that needs to be solved, for providing guidance about the possible ways to proceed and for being a reference against which the behaviour of the solutions may be judged.

There are numerous specification languages available to designers and developers, each with their strengths and weaknesses. Informal prose may help to communicate information to readers with a wide range of technical and non-technical backgrounds but they are liable to contain imprecision, omissions and contradictions. Formal specification languages based on mathematical or logical notations such as VDM [50] or Z [72] are designed to be amenable to rigorous mathematical analysis but require a certain level of expertise on the part of the reader. Omissions and inconsistencies within specifications written using a formal language may be obvious to the reader or they may become apparent when expected properties of the specification cannot be derived or proved.

As we mentioned in Section 1.3.3, the Larch [37] methodology of formal specification upon which our work is based uses a two tiered approach. Abstract specifications are written in the algebraic specification language LSL and procedural interface specifications are written in the BISL tailored to the target programming language. The emphasis is on using algebraic specifications to describe the abstractions that the solution to a problem ought to have independently of any implementation. Procedural interface specifications define the interfaces of program components using the abstractions provided by LSL specifications.

Other methodologies may have different aims and intended uses but most, if not all, can be used to help reduce the chance of mistakes and errors appearing in programs. By clearly and/or unambiguously defining the problem and/or properties of its solution, possible flaws in the design may be detected early on before an implementation is produced (by which time it may be very expensive to rectify). The implementation itself can be constructed bearing in mind the abstractions and properties defined by the specification and hopefully the finished product can be checked that to see if it satisfies them.

2.2.3 Reification

Reification [50] is the process of repeatedly applying transformations to a abstract specification to obtain a new specification. The usual objective is to transform an abstract specification of a problem into a concrete specification which closely resembles a program that could be executed and is related to the programming technique of stepwise refinement. Each transformation must be justified: a *retrieve* function must be defined that will reverse the transformation and thus show that no information has been lost. This technique allows abstract specifications which were constructed during the design stage to play a direct *rôle* in implementation process. If each reification step can be justified and the concrete specification that is finally produced is close enough to a program in the target programming language then one can argue that there are many fewer opportunities for bugs to creep in. However, the strength of this argument relies on the validity of each transformation.

In Chapter 3 we investigate the use of reification as applied to a quantum mechanics problem which is to be implemented in the object language of a computer algebra system. We show how the abstract VDM specification of the problem was obtained from a mathematical description and show that it can be implemented almost directly in **axiom**. We proceed to show that reification can be used to obtain more efficient implementations.

Outside of the VDM world, reification is central to the Extended ML [75] programming language which was mentioned in Section 1.4.1. Algebraic specifications are an integral part of the language and stepwise reification is used to transform them into suitable implementations. This is made easier by the fact that EML is a functional programming language but users are still required to formally prove each transformation.

2.2.4 Program derivation and synthesis

In the previous section we pointed out that the reliability of the program that is generated by reification depends on the reliability of the transformation steps and their justification. Since the source of the unreliability is likely to be the human who is conducting the reification process it would seem sensible to make use of a machine where it is possible.

Program synthesis is a method for creating programs from proofs in a constructive logic that the specification of the program is satisfied for all inputs and outputs. The construction that is generated as an integral part of the proof is extracted and expressed as an executable program. This technique benefits considerably from automated theorem proving systems and has related applications in program transformation. For example, in [41] Hesketh *et al* show how tail-recursive programs can be automatically synthesised from a specification. However, we are uncertain whether the efficiency of the generated code would allow this technique to scale up to large programs.

2.2.5 Cleanroom

The Cleanroom approach [21] to software development is also worthy of note. The motivation is based on the clean rooms used by silicon chip manufacturers where the cost of removing defects from chips is much more expensive than preventing them occurring in the first place. The Cleanroom methodology endeavours to ensure that the programs submitted for testing are free from errors, thereby allowing the testing to certify the reliability of a component rather than locating mistakes in it. Some of the benefits are likely to arise from careful management of developers, strict adherence to the development methodology and the continual review of components of the system to determine their correctness and quality. However, reification is utilised to obtain implementations from specifications.

2.3 Error detection

It is not always possible to construct programs which can be shown to satisfy their specification and which are guaranteed to be free from errors. Thus we need techniques to help the developer detect as many problems as quickly and easily as possible. In the sections that follow we look at a number of different techniques and comment on their suitability.

2.3.1 Runtime assertions

The simplest technique for detecting mistakes in programs is probably the use of runtime assertions. This is generally achieved by using a function that takes a boolean argument and aborts the program if the argument is false. Languages such as C provide this function as part of the standard library while in others, such as Eiffel [65], it is part of the language. If such a function is not available it is often trivial to implement.

Statements of the form `assert(`*`predicate`*`)` are inserted at various points in the program which is then tested appropriately. If the program terminates due to a failed assertion one can only assume that there is a mistake in the program or that the assertion was incorrect. The drawback is that we cannot easily tell where the error occurred in the program unless the assertions are very simple or the assertion checker explains the failure—an assertion that consists of many conjoined predicates will be of little help unless the user can identify which predicates are false. The assertions may need to provide adequate coverage of the area in which the bug arose to enable the source of the problem to be precisely located. Even if the program runs without any assertions failing it does not necessarily mean that there are no mistakes in the program.

The approach adopted by the Eiffel language is to consider violations of the pre-conditions as an error that may be caught by the exception handling system. This means that the programmer has some control over how invocations of a routine in invalid states may be dealt with and leads to more robust software.

It is important to note that there are limitations to the expressiveness of an assertion language. For example, while many procedural interface specifications (see Section 2.3.6) can be translated into runtime checks, those involving quantifiers are likely to present significant problems. It is fair to say that most runtime systems do not have the theorem

proving support to check quantified statements and the enumeration of all possible states (*c.f.* model checking) may be infeasible. However, the APP tool for C programs [74] shows how effective assertions can be when combined with procedural interface specifications.

2.3.2 Syntax checking

At the heart of all compilers is a syntax checker [2] which is used to ensure that the source code being translated conforms to the rules defining what it is to be a legal program in the input language. This kind of analysis is often the easiest to make, particularly if the grammar of the language is unambiguous and has been created with a compiler in mind.

The types of programming error that can be detected with this technique are those involving typing mistakes such as extraneous or missing characters, mis-spelt keywords and unbalanced parentheses. However, the design of the programming language can help the syntax checker perform better analysis by increasing the context sensitivity of the language. One way of achieving this is to include redundant keywords. For example, in many languages the expression used in an **if** test must be followed by the **then** keyword. The latter adds nothing to the language, except perhaps to make it more readable, but it may help to identify errors such as malformed expressions which are detected when **then** appears to be used in the wrong place. Without this the checker may not be able to distinguish the **if** test from the **if** body and generate confusing error messages.

2.3.3 Type checking

Although untyped languages can provide a large amount of flexibility they also give the programmer a greater opportunity to make mistakes through incorrect typing. Typed languages allow the programmer to provide more information about the (intended) meaning of identifiers and values. Not only can this information be used to help a human reader it can be utilised by the compiler to perform extra checks on programs and might even be used to improve performance. For example, if a function was written under the assumption that one of its arguments would always be a positive integer, then in an untyped language the programmer must remember to check that this restriction is satisfied. In a typed language it ought to be possible to define a new type that represents positive integers and annotate

the function parameters with this type. A type checker is used to decide whether or not this function is being invoked with values of the correct type and inform the programmer if this is not the case.

Unfortunately explicitly annotating every symbol with a type is both tedious and likely to make the program much harder to read. As a result, compilers for strongly-typed languages often include a type-inference engine as part of the type checker. The type-inferrer uses any information that can be found in the program to try to deduce the types of all identifiers. If it is able to find a unique type for each symbol then the program is considered to be type-correct and the types that were inferred are used as if the user had supplied them explicitly. If the type-inference engine is unable to find any suitable type or finds more than one possible type for a given symbol then an error message is displayed. Such error messages either indicate a mistake in the program or that the programmer needs to supply extra type information to resolve ambiguities.

Ideally a type checker ought to be able to reject all programs which are not type correct but this may result in some programs being rejected when they are in fact correct. For example, the following Aldor program could be regarded as being statically type-correct since the **else** branch can never be executed. However, it will not compile since the Aldor compiler assumes that it is possible for either of the branches to be executed.

```
local var:Integer;

if (true) then
   var := 42;
else
   var := "Hi there!";
```

Both **axiom** and Aldor have a rich and powerful type system that is based on a two-level object model of categories and domains. Categories are similar to Haskell [44] type classes and can be regarded as interfaces to domains; domains may represent abstract data-types or collections of functions and data. Categories and domains may be parameterised by arbitrary values, usually types or functions, which increases the expressiveness of the language. For example, it is possible to define a domain `Matrix` which is parameterised by the type of its elements `R`, where `R` satisfies the category of `Ring`. Informally, this means that the compiler will only allow matrices over a ring to be constructed which eliminates the chance that an invalid kind of matrix will be used (assuming that a domain which satisfies `Ring`

actually corresponds to the mathematical notion of a ring). It also means that the implementation of `Matrix(R)` knows what operations the domain `R` provides and can rely on their properties. For example, the addition operator `+` can be assumed to be commutative even though the compiler cannot verify this.

Another interesting feature of the Aldor type system is the provision of dependent types. This facility can be used to provide parametric polymorphism [11] and to allow values to be returned along with a type context. Below is an example of dependent types:

```
reduce(T:Type, l:List(T), op: (T, T) -> T, start:T): T ==
{
   -- Reduce the list 'l' using the operation 'op'. The
   -- starting value of the reduction is 'start'. Eg:
   --    listSum := reduce(Integer, aList, +, 0);
   local result:T := start;

   for x in l repeat
      result := op(result, x);

   result;
}
```

Here the types of the second, third and fourth arguments of `reduce` as well as the return value, all depend on the value of the first argument. Mutually dependent types are permitted although there are other limitations that are not found in functional programming languages with dependent types. These mainly arise because the compiler does not evaluate type expressions at compile time and thus the type represented by the integer 5 is considered to be different from the type represented by the integer expression $2 + 3$. Thompson and Poll [70] have proposed to rectify this problem by extending the Aldor type system so that types may be evaluated at compile time.

2.3.4 Data and control-flow analysis

Certain classes of programming mistakes such as data and control-flow anomalies can be detected with static program analysis. Both types of analysis divide the program into sections called *basic blocks* [2]. These are linear sequences of statements with no jumps or branches in them. We consider functions that can never terminate and statements that can never be executed as control-flow anomalies. Data-flow anomalies include:

- use of variables before they have been assigned well-defined values. For example, the failure to initialise an `Integer` variable in Aldor or a pointer in C may produce undesirable runtime behaviour.
- variables that are declared but not used. This may indicate that other variables have been used incorrectly in their place.
- consecutive assignments to a variable without any intervening reads. This may be because statements between the assignments have been omitted or that one or more of the assignments is to the wrong variable.

Static program analysis may be performed locally or globally and each has its benefits and drawbacks. Global program analysis may provide a complete picture of the entire program but it can easily become computationally expensive and may be infeasible for large programs. On the other hand, local program analysis examines a few basic blocks at a time and provides a conservative estimate of the global view. Although it may miss anomalies that are detectable by global analysis, its complexity is related to the number of basic blocks examined at each step rather than the size of the whole program. Thus local program analysis is usually adopted and applied to each function or procedure in turn.

One of the first tools to detect data-flow anomalies was DAVE [29] which was designed to analyse Fortran programs using in-built rules. Cesar [67] also worked on Fortran programs and using sequencing constraints based on regular expressions written in Cecil [67] to perform inter-procedural analysis. Since Cecil is independent of any particular programming language, Cesar can be extended to analyse programs written in other languages such as C and Ada [67]. The UNIX `lint` utility [79] for C performs stricter type-checking than the early C compilers did and more recent versions also look for common data-flow anomalies such as use-before-definition errors, dead code and failure to use return values from functions. Widespread use of C and the benefits of static checking has encouraged the developers of modern C and C++ compilers undertake this kind of analysis automatically.

Another tool based on `lint` is LcLint [25]. This also works on C programs but combines Larch/C interface specifications (see Section 4.1.1) with its own special annotation language to enable it to perform a wider range of checks than standard `lint`. Many of the checks are based around detecting memory management and pointer errors: the bane of many C programmers. Users may decide to follow an abstract data-type style of programming (abstract data-types themselves are not available in C) and use LcLint to check

for any violations of information hiding or naming conventions that have been adopted for this approach. Other checks include detecting code whose behaviour depends on the order of evaluation, code which might exceed the limits of compilers on some systems as well as checks on macros.

The Aspect [45] data-flow analysis tool can be used to detect missing dependencies between "aspects" of abstract objects. As an example, consider a procedure which searches a non-empty unsorted list of integers `L` for the smallest element `n`. Clearly all the elements of `L` must be examined, so one could specify that the value of `n` in the post-state of the procedure depends on the value of `L` in the pre-state. As with run-time assertions, the dependency checking performed by Aspect is only able to highlight dependencies which are missing from the code and is unable to pinpoint where the programmer ought to look for the mistake. However, Aspect was designed for detecting bugs in languages such as CLU where the design of the language has made many of the `lint`-style checks redundant and thus the detection of more subtle programming mistakes is more beneficial.

2.3.5 Symbolic execution

Instead of executing a program on concrete test data, symbolic data can be used to represent different classes of inputs that a program can accept. These input classes could be derived from control-flow analysis: for example, only those inputs which affect the flow-of-control of the program need to be considered.

Given a program and some symbolic input data, an "execution tree" can be constructed which represents all the paths which might be followed during execution. This tree can be used in several ways:

- it might be examined to generate concrete test data
- assertions about the output can be checked against various possible execution routes.
- user-supplied assertions and the structure and content of the tree could be used to generate verification conditions (see Section 2.3.7).

One problem faced by symbolic execution is the production of infinitely deep trees which arise from the analysis of conditional statements. In the case of the EFFIGY system [54] for PL/I, user-interaction is used to guide the traversal of the tree with support for backtracking;

when used for automatic test-case generation the user must impose a limit on the depth of the search. An alternative method might require the formalisation of induction over symbolic execution trees to permit "execution" over infinitely deep branches.

2.3.6 Procedural interface checks

A well-known specification technique that can be applied to procedural languages is based around the definition of pre- and post-conditions for functions and abstract data-type methods. Two such systems are Larch [37] and VDM-SL [50]. However, there do not appear to be many tools in existence which are able to make full use of the interface specifications when checking the source code. For example, the LcLint tool [25] is able to use Larch/C interface specifications to improve the types of static data-flow analysis that it can perform. However, it only uses the information relating to the modification of client visible state and, at the time of writing, does not make any use of the pre- and post-conditions. Inscape [69] is another system for performing procedural interface checks; it uses conditions and obligations between caller and callee to highlight mistakes in user programs. A review of different Larch interface specification languages can be found in Chapter 4.

We believe that the information provided by interface specifications can be used not only as clear and concise documentation, but also as a way of generating verification conditions to help detect programming mistakes. If the implementation of a library procedure $\mathcal{X}$ has been shown, or is assumed, to satisfy its interface specification, then it is relatively straightforward to generate verification conditions for each application of $\mathcal{X}$ to show that the pre-condition is satisfied. The information provided by the post-condition can be used to extend a database of knowledge of the program state after $\mathcal{X}$ has finished executing. This technique is discussed in more detail in Chapter 5.

2.3.7 Verification condition generation

To verify the correctness of a program, we require a set of criteria against which the program can be judged. The criteria need to be expressed in a specification language that has a well defined semantics and which is amenable to formal manipulation and proofs. In addition, a formal semantics of the programming language, or a useful subset of it, is required

so that programs can be reduced to purely logical or mathematical expressions. A popular approach is to decompose the program into loop-free segments and to attach assertions to each based on the global correctness assertion [30, 33, 42]. Using the formal semantics of the programming language the assertions are pushed back through each section of the program until a statement that the post-conditions are logical implications of the pre-conditions is obtained (the verification condition). This process is repeated for each loop-free section that corresponds to a possible execution path. An alternative method is to start from the pre-condition and push it through the section to obtain the strongest post-condition. This method suffers from the problem that the VC may contain information relating to temporary calculations that is actually irrelevant to the correctness of the program. Hence the post-condition may be much larger and more complex than is necessary [62].

The addition of loops and procedures complicate the process of VC generation since many imperative languages do not have well defined semantics of loops. Not only that but procedures may perform operations which affect the global program state: so-called side-effects. However, given sufficient care, loops may be investigated using induction techniques with the addition of a termination requirement. If the program is modular then procedures can be regarded as verified sub-programs—if the pre-condition for the invocation of the procedure holds then we can assume that the execution terminates in a state where the post-condition is true. The procedures themselves can be verified separately if necessary.

In Chapter 5 we introduce these techniques of program verification and describe our lightweight approach. Given that automatic program verification is generally undecidable [30] we believe that, at least in the context of systems involving large amounts of mathematical knowledge such as CAS, it is helpful to have a tool that can generate verification conditions and leave the user to decide what to do with them. In Chapter 5 we suggest that some verification conditions which would present significant difficulties to automatic theorem provers might be easily discharged using expert knowledge of the user. At other times, such as in non-critical situations, the user may decide to ignore some or all of the verification conditions depending on how difficult or important the proofs are. Others may simply be noted for future reference or used to place additional restrictions on the use of the program so that the verification conditions can be trivially discharged.

Two examples of verification condition generators with quite different philosophies are Penelope [35] for Ada and the Extended Static Checking System or ESC [22] for Modula-3. Penelope is an interactive syntax-directed editor for programs written using a subset of

Ada that has a formal semantics. It can automatically generate verification conditions from user-supplied assertions and Larch/Ada [35] specifications, and then attempt to discharge them. The user is able to provide simplifying lemmas and rewrite rules to assist the proof attempt, they can modify the program (which may change the set of verification conditions), or they can simply note them for future reference. Penelope can be used for full program verification but relies heavily on user interaction.

In contrast to this ESC is completely automatic and is based on the view that program checking must be fast enough that the programmers are not discouraged from using it, and it must be automatic just like type checking so as not to be tedious. To achieve these aims ESC is intended to be used for limited program verification with goals of detecting simple programming errors such as dereferencing of NIL pointers or the detection of certain types of race conditions and deadlock in concurrent programs. The user annotates their program with specifications which are transformed into verification conditions. These are passed to the SIMPLIFY [22], an automatic theorem prover specifically designed for use with ESC. If a proof attempt fails, an example contradiction to the verification condition is presented to the user. Since some valid formulae may be unprovable in this system an heuristic limit to the search depth is used to ensure termination.

2.4 Relation to Aldor and this thesis

In Section 2.1 we highlighted various problems that can affect large software systems such as the **axiom** CAS [48]. Central to each of the issues was documentation or the lack of it and that is a part of what this thesis is about. We are proposing to apply the Larch approach of two-tiered formal specification to Aldor [83] and **axiom**—in the first instance Larch/Aldor BISL specifications can be used to allow designers and developers to write clear, concise and (hopefully) unambiguous documentation which will serve as a strong basis for communication. In Section 2.2.3 we introduced the technique of reification and this is covered in more detail in Chapter 3. With reification, specifications can be successively refined to obtain concrete implementations and may also be used for optimisation. Finally in Sections 2.3.6 and 2.3.7 we looked at how interface specifications can be used by static program analysers to investigate whether or not a program fragment is being used in the correct context. In Chapter 5 we will look at various techniques for program verification and describe our ideas for lightweight verification condition generation using Larch/Aldor

interface specifications.

As we mentioned in Section 1.3.6, Aldor is a strongly-typed imperative programming language that was designed to enable computer algebra routines to be implemented in a natural and efficient way. It is possible to provide extensions to **axiom** which are implemented in Aldor and since this language can offer benefits to other developers we feel that it is is better to concentrate on Aldor rather than the object language of **axiom**. Recent developments with the Maple CAS also hint that Aldor may be used more widely than just in the **axiom** community and thus we believe that the potential benefits of Larch/Aldor are bigger than those of Larch/**axiom**.

Chapter 3

Reification for computer algebra systems—a case study

In this chapter we investigate how the VDM reification [50] techniques can be applied to programs written for computer algebra systems. In particular we use reification as a way of obtaining more efficient (faster) implementations of an **axiom** program designed to compute relative atomic oscillator strengths of hydrogen-like atoms. We also use it to help select a pragmatic representation of the real numbers suitable for this case study, and to investigate the use of interface specifications and verification techniques for program development. Limitations of computer algebra systems are also touched upon.

Reification is the process by which one specification is transformed into another with the provision of a justification for the transformation. Usually one starts with an abstract specification of a problem which is transformed into a specification that is closer to the chosen implementation language. This process is repeated as many times as necessary. Reification can be split into two separate processes—data reification and operation decomposition. Data reification is used to transform abstract data types in the specification into concrete data types that are closer to those of the target programming language. Operation decomposition is used to develop implementations from abstract operators which appear in the specification. The requirement of justifications for these transformations gives the user increased confidence in the reliability of the results. In addition the process of making these justifications forces the developer to think very carefully about the choices that are made at each stage. We believe that this helps to improve the quality of the implementation.

The task of computing hydrogenic oscillator strengths is an interesting one. Given the facilities of computer algebra systems such as **axiom** it is actually possible to implement abstract specifications of the problem directly. This allows us to measure quantatively the effect of transforming one specification into another. In this case study we concentrate on the use of data reification to provide a suitable implementation of the real numbers and other concrete types required for the computation. The choice of a data type to represent the real numbers required us to circumvent the type system of **axiom** to a certain degree. To regain the safety lost by this approach we rely on interface specifications and the reification arguments justifying the choice. Operation decomposition is used to obtain implementations that are more efficient than their predecessors, *i.e.* they are able to compute the solution to the problem in less time.

We begin in Section 3.1 by describing the real-world quantum mechanics problem that we are tackling and provide the equations needed to solve it. Then in Section 3.2 we give VDM specifications and use data and operator reification to obtain an implementation closely related to the abstract specifications. After investigating whether the implementation satisfies its specification we repeat the reification process to obtain successively more efficient implementations. During these stages we ask whether the implementations satisfy their specifications, and whether or not they satisfy the previous implementation; an alternative reification path is also described in Section 3.4. We find that most of the work for operator reification actually takes place before the VDM specifications are written. In Section 3.1.2 we make use of standard algebraic manipulations to simplify the expressions which we wish to compute with: in the context of this case study these transformations represent operator decompositions and we (implicitly) appeal to undergraduate mathematics for their justification. Thus the approach to this problem, which might be adopted by any physicist, can naturally be thought of as reification.

Finally in Section 3.5 we discuss the issues raised. These include the implementation of real numbers in computer algebra systems such as **axiom** and definite integration: how well do computer algebra systems cope with this task and how reliable are their results? We consider the uses of formal methods and briefly touch on the subject of method selection by way of specification matching.

3.1 Hydrogenic oscillator strengths

In this section we describe the mathematics behind the atomic oscillator problem in preparation for the various implementations. The aim is to construct **axiom** programs that can compute the weighted mean strength of transitions in hydrogen atoms from one atomic level to another. These programs are the result of reifying Equation 3.14 which defines this quantity. The reification process is described in Section 3.1.2 and generates three equations representing different formalisations of the problem:

$$\begin{aligned} r_{ab} &= \int_0^\infty R_{n_b l_b}(r)\, r\, R_{n_a l_a}(r)\, r^2\, dr \\ r_{ab} &= \int_0^\infty Q_{n_b l_b}(r)\, Q_{n_a l_a}(r)\, e^{-\left(\frac{n_a+n_b}{n_a n_b}\right) r}\, r^3\, dr \\ r_{ab} &= \sum_{i=0}^{\beta} \frac{c_i\, i!}{\alpha^{(i+1)}} \end{aligned}$$

These equations form the basis of the VDM specifications described in subsequent sections, and we appeal to standard techniques of algebraic manipulation for the justification of the reification from one to the other. Although this section necessarily contains some quantum mechanics, readers with a basic knowledge of integration ought to be able to follow the reasoning even if they do not understand the meaning of the equations. A more thorough description of the problem and the mathematics behind it can be found in texts such as [6].

3.1.1 Weighted mean line strength

The aim of this study is to compute the weighted mean line strengths of transitions from one atomic level n_a to another n_b for hydrogen atoms. It can be easily generalised to other hydrogen-like atoms by introducing their atomic mass Z (for hydrogen $Z = 1$). Each atomic level n is subdivided into a number of different states of equal energy where each state is identified by spin and magnetic quantum numbers l and m respectively. The quantum numbers n, l and m all satisfy

$$n \in \{1, 2, 3, \ldots, \infty\} \tag{3.1}$$

$$l \in \{0, 1, \ldots, n-1\} \tag{3.2}$$

$$m \in \{-l, -l+1, \ldots, 0, \ldots, l-1, l\} \tag{3.3}$$

The number of states sharing the same energy is called the degeneracy. We define the degeneracy of states that share the same values of n and l as g_{nl} and the total degeneracy of atomic level n as g_n. It follows that

$$g_{nl} = 2l + 1 \tag{3.4}$$

$$g_n = \sum_{l=0}^{n-1} g_{nl} = 2n^2 \tag{3.5}$$

From quantum mechanics it is known that the weighted mean line strength can be described by the equation

$$f_{n_a n_b} = \frac{1}{g_{n_a}} \sum_{l_a} \sum_{l_b} \left(\frac{2 l_{\max} \delta_{l_a l_b} E_{ab}}{3} \right) |r_{ab}|^2 \tag{3.6}$$

where E_{ab} is the energy difference between states $a = (n_a, l_a, m_a)$ and $b = (n_b, l_b, m_b)$:

$$E_{ab} = E_a - E_b = \frac{1}{n_a^2} - \frac{1}{n_b^2} \tag{3.7}$$

The terms $l_{\max}$ and $\delta_{l_a l_b}$ represent selection rules which arise because transitions from one arbitrary state to another may be forbidden by quantum mechanics. The rules for the most common type of transition (electric dipole) require that $\Delta l = \pm 1$ and either $\Delta m = 0$ or $\Delta m = \pm 1$ where $\Delta l = l_a - l_b$ and $\Delta m = m_a - m_b$.

$$l_{\max} = \max(l_a, l_b) \tag{3.8}$$

$$\delta_{l_a l_b} = \begin{cases} 1 & \text{if } |l_a - l_b| = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

The remaining term in Equation 3.6 is r_{ab} which represents the radial component of an equation not described here (the dipole transition matrix). We note in passing that the dipole transition matrix is defined using spherical polar coordinates and the angular components give rise to the selection rules described above. It can be shown that

$$r_{ab} = \int_0^\infty R_{n_b l_b}(r)\, r\, R_{n_a l_a}(r)\, r^2\, dr \tag{3.10}$$

where

$$R_{nl}(r) = N_{nl} e^{-\frac{\rho}{2}} \rho^l L_{n+l}^{2l+1}(\rho) \tag{3.11}$$

$$N_{nl} = \sqrt{\left(\frac{\rho}{r}\right)^3 \left[\frac{(n-l-1)!}{2n\,(n+l)!^3}\right]} \tag{3.12}$$

and where $\rho = \frac{2r}{n}$. The associated Laguerre polynomial, L_p^q, is defined by:

$$L_p^q(x) = (-1)^q \sum_{s=0}^{p-q} \frac{(-1)^s\,(p!)^2}{s!\;(q+s)!\;(p-q-s)!}\, x^s \tag{3.13}$$

3.1.2 Solving the integral

Equation 3.6 can be rewritten using equations from the previous section to give

$$f_{n_a n_b} = \left(\frac{n_b^2 - n_a^2}{3 n_b^2 n_a^4}\right) \sum_{l_a=0}^{n_a-1} \sum_{l_b=0}^{n_b-1} \delta_{l_a l_b}\, l_{max}\, r_{ab}^2 \tag{3.14}$$

This is clearly easy to evaluate for given values of n_a and n_b once the value of r_{ab} is known. However, Equation 3.10 which defines r_{ab} cannot be implemented very easily in many programming languages since generally they do not have the libraries that are essential for representing polynomials and do not provide the necessary integration facilities.

In this section we will transform Equation 3.10 to obtain an equation for r_{ab} that can be easily implemented in programming languages such as C or Fortran (Equation 3.21). In doing so we will also obtain other intermediate equations for r_{ab} which can be evaluated by **axiom** with varying degrees of speed. Using the substitution

$$R_{nl}(r) = Q_{nl}(r)\, e^{-\left(\frac{r}{n}\right)} \tag{3.15}$$

with Equations 3.11 and 3.12 we can rewrite Equation 3.10 as

$$r_{ab} = \int_0^\infty Q_{n_b l_b}(r)\, Q_{n_a l_a}(r)\, e^{-\left(\frac{n_a+n_b}{n_a n_b}\right) r}\, r^3\, dr \tag{3.16}$$

This equation can be solved by **axiom** and will be used in Section 3.3 as the basis of the first reified specification. We note that this is a Laplace transform of the product of the polynomials $Q_{n_a l_a}$, $Q_{n_b l_b}$ and r^3 and, after using a little algebra (operation decomposition), we find that

$$Q_{nl}(r) = r^l \left(\frac{2}{n}\right)^{\left(\frac{2l+3}{2}\right)} \sqrt{\frac{(n-l-1)!}{2n\,(n+l)!^3}}\, L_{n+l}^{2l+1}\left(\frac{2r}{n}\right) \tag{3.17}$$

Now we have an equation for r_{ab} consisting of the monomial r^3, an exponential of the form $e^{-\alpha r}$ and the product of two polynomials in r where the coefficients of the polynomials are in terms of n and l only. Since $Q_{nl}(r)$ is a polynomial with a finite number of terms (see Equations 3.13 and 3.17), $Q_a\, Q_b\, r^3$ can be expressed as

$$Q_a\, Q_b\, r^3 = c_0 + c_1 r^1 + c_2 r^2 + \cdots + c_n r^n \tag{3.18}$$

where $n = (n_a + n_b) - (l_a + l_b) + 1$ and where Q_a is a shorthand for $Q_{n_a l_a}(r)$. This means that Equation 3.16 can be written as a sum of integrals:

$$r_{ab} = \left[\int_0^\infty c_0\, e^{-\alpha r} dr + \int_0^\infty c_1 r\, e^{-\alpha r} dr + \cdots + \int_0^\infty c_n r^n\, e^{-\alpha r} dr\right] \tag{3.19}$$

Now since

$$\int_0^\infty r^n e^{-\alpha r} dr = \frac{n!}{\alpha^{(n+1)}} \tag{3.20}$$

we can rewrite Equation 3.19 to give

$$r_{ab} = \sum_{i=0}^{\beta} \frac{c_i\, i!}{\alpha^{(i+1)}} \tag{3.21}$$

where c_i is the coefficient of r^i of the polynomial $Q_a\, Q_b\, r^3$ and where

$$\alpha = \frac{n_a + n_b}{n_a\, n_b} \tag{3.22}$$

$$\beta = (n_a + n_b) - (l_a + l_b) + 1 \tag{3.23}$$

This equation is used in Section 3.3.3 as the basis of the most concrete specification. After a little inspection one can see that it would be easy to write a program in a language such as C or Fortran that can evaluate Equation 3.14 using Equation 3.21.

3.1.3 Symbolic mathematics using a computer algebra system

In the previous sections we have applied the techniques of algebraic manipulation to transform Equation 3.10 into Equation 3.21 and, as we shall see later on in this chapter, **axiom** is able to evaluate the latter much more quickly than the former. In addition the latter equation can easily be implemented in a programming language such as C or Fortran whereas the former presents significant problems.

The reader may be wondering why we have not discussed the use of a computer algebra system to assist with the algebraic manipulations of Section 3.1.2, particularly since this field is often regarded as the *forte* of such systems. Unfortunately the reason is that computer algebra systems may be unable to perform the symbolic mathematics that we need and this is illustrated with an example from this case study.

Equation 3.16 may seem daunting at first so it is natural to undertake further investigations of it using a computer algebra system. However, the presence of the seemingly innocuous parameters n_a and n_b may place the equation beyond the scope of the chosen system. The result is that the computer algebra system may simply refuse to evaluate the expression or it may attempt to do so but return an invalid or untrustworthy result.

For example, if we ask **axiom** to evaluate $\int_0^\infty x^n \exp(-\alpha x)dx$ it returns the result "failed" which indicates that it was unable to answer our request—the equation is beyond the scope of the `integrate` function. If specific values for n and α are provided then a result can be obtained. Requesting the indefinite integral for a specific n (and arbitrary α) does return a suitable answer but this is of no help with the definite integral shown. Similarly Maple V is unable to provide a solution to the equation, even when assumptions are placed on the types of the symbols n and α. However, Mathematica 2.0 succeeds returning the answer

$$\frac{\Gamma(n+1)}{\alpha^{n+1}} \tag{3.24}$$

which, although it is correct, leaves us wondering what assumptions were made on the type of n and α for this result to be reached. In principle these symbols could represent values from a variety of types which would render the equation meaningless. Perhaps recognising that the equation represents a Laplace transform may help?

$$\mathcal{L}\{f(x)\} = \int_0^\infty f(x)\exp(-sx)dx$$

Unfortunately this is still out-with the scope of **axiom** unless we provide specific values for n. Maple V performs better returning the same result as Mathematica did without needing any assumptions on n.

Thus at one extreme **axiom** will not provide a solution to the integral as it stands, presumably because it does not have any information about the type of the parameters n and α. At the other extreme Mathematica 2.0 returns the correct solution but this is tempered by the fact that it has made unspecified assumptions about the parameters; Maple V is only able to produce a solution to the problem when specifically asked to compute the Laplace transform. As we expect, computer algebra systems are not omnipotent and while they may give correct answers to the majority of questions put to them we must be wary. When using them we must be careful to note any assumptions that they make and to check their results whenever possible. There will be times, as shown here, when our chosen system cannot give us the answer to a particular problem and we will need to look to another source—either a different system or a suitable reference text—for the further assistance.

3.1.4 Summary

In the previous two sections we obtained an expression for the weighted mean line strength of electric dipole transitions between atomic levels n_a and n_b (Equation 3.14). We have solved the radial integral for r_{ab} and in doing so we obtained three different expressions for it (Equations 3.10, 3.16 and 3.21). This process represents the majority of the work of operation decomposition described in more details in the following sections. These equations form the basis of the specification of the problem; Equation 3.10 represents the abstract specification which has been successively refined or reified to obtain a more concrete specification in the form of Equation 3.21. Although the abstract specification of Equation 3.10 cannot be easily implemented in a programming language such C, it could be implemented in a language such as those used by computer algebra systems. We will utilise this fact in the next section to provide quantative comparisons of the effect of reification.

We have also noted that while computer algebra systems can be extremely valuable for solving problems which are beyond the pen-and-paper approach, they are not a panacea and need to be used with a certain degree of caution. Although we were unable to use **axiom** to help with all the symbolic computation of Sections 3.1.1–3.1.2, we will see that it is extremely good at computing the results of our equations for specific values of parameters such as n_a.

3.2 Implementing the abstract specification

In the previous section we derived an equation for the weighted mean strength of a hydrogen-like atomic oscillator (Equation 3.14). We applied reification techniques in the form of operator decomposition to obtain three further equations, 3.10, 3.16 and 3.21 which represent different levels of abstraction. The standard techniques of algebraic manipulation provide the justification for each reification step and will not be covered here. In this section we use Equations 3.10 and 3.14 as the abstract specification of our problem, produce VDM interface specifications and then construct an implementation in **axiom**. The functions which appear in these equations are used as the basis for the functions in our implementation, naturally modeling the original mathematical specification.

3.2.1 From abstract specification to interface specification

We can write the VDM style specification of an **axiom** function to compute the value of Equation 3.14 for specific values of n_a and n_b simply as follows:

$$f_{n_a n_b} \triangleq \left(\frac{n_b^2 - n_a^2}{3 n_b^2 n_a^4}\right) \sum_{l_a=0}^{n_a-1} \sum_{l_b=0}^{n_b-1} \delta_{l_a l_b}\, l_{max}\, r_{ab}^2$$

$$\begin{aligned} &\texttt{Fab}\ (n_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z})\ z : \boldsymbol{R} \\ &\quad \textbf{pre} \quad (n_a > 0) \wedge (n_b > 0) \\ &\quad \textbf{post} \quad z = f_{n_a n_b} \end{aligned}$$

The first line defines an equation which represents the functional specification of the interface and which is used in the post-condition of behavioural interface specification which follows it. The latter can be considered as a template of the function `Fab` which is being implemented. The arguments of the function are given in parenthesis after its name using the syntax of a general programming language and the types of the VDM-SL specification language. The symbol z appearing after the parameter list of the function represents the value returned by the function and allows such a value to be referred to in the behavioural interface specification. Other specification languages such as the Larch family [37] use a pre-defined symbol such as *result* instead. The only restriction on this function is that the atomic level numbers n_a and n_b must be positive integers. The definitions of $\delta_{l_a l_b}$ and l_{max} are trivial (see Section 3.1.1 and Equation 3.9) and need not be specified. The specification of r_{ab}^2 follows from Equation 3.10:

$$r_{ab} \triangleq \int_0^\infty R_{n_a l_a}(r)\, R_{n_b l_b}(r)\, r^3\, dr$$

$$\begin{aligned} &\texttt{Rab}\ (n_a : \boldsymbol{Z},\ l_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z},\ l_b : \boldsymbol{Z})\ z : \boldsymbol{R} \\ &\quad \textbf{post} \quad z = r_{ab} \end{aligned}$$

which should be self-explanatory; note that there are no pre-conditions. Likewise

$$R_{nl}(r) \triangleq N_{nl} \left(\frac{2r}{n}\right)^l e^{-(r/n)}\, L_{n+l}^{2l+1}\left(\frac{2r}{n}\right)$$

$$\begin{aligned} &\texttt{Rnl}\ (n : \boldsymbol{Z},\ l : \boldsymbol{Z})\ z : \boldsymbol{R}\,[x] \\ &\quad \textbf{pre} \quad (n \neq 0) \\ &\quad \textbf{post} \quad z = R_{nl}(r) \end{aligned}$$

where $\boldsymbol{R}[x]$ represents the polynomials over the real numbers.

A definition of the associated Laguerre polynomial $L_{n+l}^{2l+1}\left(\frac{2r}{n}\right)$ can be found in the form of Equation 3.13 and, since it is provided as an **axiom** library function, there is no need to specify it here. All that remains is to specify a function to compute N_{nl}

$$N_{nl} \triangleq \sqrt{\left(\frac{2}{n}\right)^3 \left[\frac{(n-l-1)!}{2n(n+l)!^3}\right]}$$

$$\begin{aligned}
&\texttt{Nnl}\ (n : \boldsymbol{Z},\ l : \boldsymbol{Z})\ z : \boldsymbol{R} \\
&\quad \textbf{pre} \quad (n - l \geq 1) \wedge (n + l \geq 0) \wedge (n \neq 0) \\
&\quad \textbf{post} \quad z = N_{nl}
\end{aligned}$$

The pre-condition for N_{nl} ensures that N_{nl} is a real-valued function. From the definition of the possible range of values for n and l in Section 3.1.1 we note that this is valid for all atomic states (n, l).

3.2.2 Constructing an implementation

With appropriate choice of concrete data types the specifications in the previous section can be implemented almost directly. The main problem is in choosing a concrete data type to represent the real numbers and polynomials over the reals. With languages such as C and Fortran the common choice is the floating-point data type, since these languages do not offer much else without significant extra work on the part of the implementer. Although **axiom** has support for floating-point calculations to an arbitrary precision we would like to do better—floating-point numbers are only an approximation to the reals and one would expect a computer algebra system to provide other options. Such an approximation would also complicate any reasoning about the implementations since the errors at each stage of the calculation must be taken into account.

Instead we have chosen to use the `Expression(R)` domain. This can be used to represent expressions involving symbolic functions with coefficients taken from a totally ordered set `R`. A variety of symbolic functions are available including the trigonometric functions $\sin, \cos$ *etc.* and special functions such as the Euler Gamma function and Bessel functions. Values of this domain can be manipulated in the same way as values of other domains such

as polynomials and may be integrated which is important for this case study. The specifications in the previous section show that all the symbols have integer coefficients. Since the integers form a totally ordered set the domain `Expression(Integer)` is an ideal choice to represent the reals in **axiom**. Although it is unlikely that this type can represent all real numbers, it can be shown to represent the subset of real numbers used in this case study (see Equations 3.14 and 3.21).

It is important to note that we cannot simply replace every occurrence of the real numbers with the **axiom** type `Expression(Integer)` when we construct our implementation. For example, the abstract type of R_{nl} is a univariate polynomial over the real numbers and so one might be tempted to use the `UnivariatePolynomial(x, R)` constructor with `Expression(Integer)` for `R`. Unfortunately this is not possible due to the type constraint on this operator that requires `R` to be a ring (`Expression(Integer)` is not). However, a univariate polynomial over the reals can be considered as an expression involving a symbol (usually x) raised to integer powers with real coefficients. Careful examination of the definition of $R_{nl}(r)$ and its constituent terms shows that it can be represented using `Expression(Integer)` directly.

The choice of types for other identifiers is more straightforward. The pre-condition of `Fab` requires that its two parameters n_a and n_b are positive integers; this is captured by the use of the **axiom** type `PositiveInteger` (abbreviated as `PI`). Similarly the second argument of `Rnl` corresponds to the quantum number l which must be a non-negative integer and so `NNI` is used. We note in passing that if the **axiom** type system was extended in the way suggested by Poll [70], it would be possible to use our interface specifications as types. This would reduce the chance that the functions we have defined could be used with invalid arguments. For example, the pre-condition of the function `Nnl` cannot be captured using the current type system of **axiom**.

The source code for this implementation can be found in Appendix B.1.

3.2.3 Checking for satisfaction

Does the implementation given in Appendix B.1 satisfy the specifications in the previous section? Since the specification consists of algebraic formulae and the **axiom** language in which the program is written is designed to implement such formulae in an intuitive man-

ner, it is possible to show that the implementation satisfies the specification by inspection assuming that the semantics of **axiom** are the same as those in the specification.

All the functions defined in Section 3.2.1 are a direct translation of the specification and therefore will be considered to satisfy their specifications. The pre-conditions of these functions are trivially discharged using the pre-condition of `Fab` and the definition of $f_{n_a n_b}$. However, two parts of `Fab` require closer examination. The first is that we have expressed the double summation as a nested iteration (a "`for`" loop) and the second is that the Kronecker delta function is implemented using an `if-then` construct. Both are common programming techniques but need to be verified before the implementation can be considered to be correct with respect to its specification.

Proof of summation implementation

We wish to prove that the **axiom** code

```
r := 0

for i in a..b repeat
   r := r + f(i)
```

is a correct implementation of the formal sum $\sum_{i=a}^{b} f(i)$ where $a <= b$. To do this we need to establish a loop invariant, that is a predicate that is true after each execution of the body of the loop and is also true when the loop terminates. This can be derived from the post-condition of the loop [23], the specification of which is

$$\begin{array}{ll} \textbf{pre} & (a \leq b) \wedge (\mathtt{r} = 0) \\ \textbf{post} & \mathtt{r} = \sum_{j=a}^{b} f(j) \wedge (\mathtt{i} = b) \\ \textbf{inv} & \mathtt{r} = \sum_{j=a}^{\mathtt{i}} f(j) \wedge (a \leq \mathtt{i} \leq b) \end{array}$$

(note that we use j to represent the summation variable in the abstract specification to distinguish it from the program variable `i` used for the same purpose in the implementation). The proof is by induction and the hypothesis to be proved is:

$$P_k : \mathtt{r} = \sum_{j=a}^{b} f(j) \tag{3.25}$$

where $k = b - a$.

1. *Base step*—establish P_k for $k = 0$ i.e. $b = a$.

   From the source code it can be seen that for the LHS of Equation 3.25 we have:

   $$\mathtt{r} = 0 + f(a)$$

   and for the RHS of Equation 3.25:

   $$\sum_{j=a}^{a} f(j) = f(a)$$

   Hence P_0 is true.

2. *Induction step*—assuming P_k is true, show that P_{k+1} is true. To do this we consider the case where we have executed the loop body with $\mathtt{i} = c + 1$ for $a \leq c < b$. From P_k we know that $\mathtt{r} = \sum_{j=a}^{c} f(j)$ and so the LHS of P_{k+1} becomes:

   $$\begin{aligned} \mathtt{r}' &= \mathtt{r} + f(c+1) \\ &= f(c+1) + \sum_{j=a}^{c} f(j) \end{aligned}$$

   Similarly, the RHS becomes:

   $$\sum_{j=a}^{c+1} f(j) = f(c+1) + \sum_{j=a}^{c} f(j)$$

Hence P_{k+1} is true and by the induction principle, P_k is true for all k. Thus the implementation of the summation shown is correct (assuming that addition is commutative) and it is easy to extend the proof to show that the double summation used in the implementation of $f_{n_a n_b}$ is also correct. The semantics of the **axiom** "`for`" loop ensures termination provided that $f(j)$ terminates for all $j \in \{a \ldots b\}$.

Proof of $\delta_{l_a l_b}$ implementation

We need to show that

```
if (abs(la - lb) = 1) then
   rab := Rab(na,la,nb,lb)
   r   := r + max(la,lb)*rab**2
```

is the same as $\texttt{r} := \texttt{r} + \delta_{l_a l_b}\, l_{max}\, r_{ab}^2$ within the double summation of Equation 3.14 where $l_{max} = max(l_a, l_b)$. The code fragment above has the effect of adding $l_{max}\, r_{ab}^2$ to `r` when $|l_a - l_b| = 1$ and adding zero at all other times. Thus it defines the function $f(l_b)$ (which corresponds to $f(j)$ used in the proof of the formal sum implementation above) as follows

$$f(l_b) = \begin{cases} l_{max}\, r_{ab}^2 & \text{when } |l_a - l_b| = 1 \\ 0 & \text{otherwise} \end{cases}$$

From Equations 3.9 and 3.14 we obtain exactly the same expression for $f(l_b)$ and so the implementation of the Kronecker delta function as an "`if-then`" construct is correct.

3.2.4 Summary

In Section 3.2.1 we translated the two main equations of the problem into VDM functional and behavioural interface specifications. These were naturally broken down using the functions appearing in these equations. However, before an implementation could be constructed we needed to apply data reification to select an appropriate **axiom** concrete data type to represent the real numbers and polynomials over the reals. In Section 3.2.2 we decided that the use of arbitrary precision floating-point numbers was not the best choice and settled on the **axiom** domain `Expression(Integer)`. The result is that Equation 3.14 can be solved for particular values of n_a and n_b with no loss of precision. Only when the values are displayed as floating-point numbers will any rounding take place. Finally in Section 3.2.3 we applied simple verification techniques to check that the implementation was correct with respect to its specification.

3.3 Towards more efficient implementations

In this section the specifications of Section 3.2 are reified to obtain new specifications and two, hopefully more efficient (faster) implementations. Most of the work of operation decomposition has already been performed in Section 3.1.2 and Section 3.2.2 has solved most of the problems of data reification. We find that we cannot provide a justification for the data reification which produced the implementation based on Equation 3.21 and discuss the issues involved. In addition we ask whether the new implementations satisfy the first implementation.

3.3.1 Reification and implementation

Examination of the specifications of Section 3.2 show that there is little that can be done with $f_{n_a n_b}$. However, from Section 3.1.2 we can reify r_{ab} from Equation 3.10 to Equation 3.16 giving a new specification:

$$r_{ab} \triangleq \int_0^\infty Q_{n_a l_a}(r)\, Q_{n_b l_b}(r)\, e^{-\left(\frac{n_a+n_b}{n_a n_b}\right) r}\, r^3\, dr$$

$\texttt{Rab}\ (n_a : \boldsymbol{Z},\ l_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z},\ l_b : \boldsymbol{Z})\ z : \boldsymbol{R}$
$\quad \textbf{pre} \quad (n_a \neq 0) \wedge (n_b \neq 0)$
$\quad \textbf{post} \quad z = r_{ab}$

Now we only need a specification of Q_{nl}:

$$Q_{nl}(r) \triangleq r^l \left(\frac{2}{n}\right)^{\left(\frac{2l+3}{2}\right)} \sqrt{\frac{(n-l-1)!}{2n(n+l)!^3}}\, L_{n+l}^{2l+1}\left(\frac{2r}{n}\right)$$

$\texttt{Qnl}\ (n : \boldsymbol{Z},\ l : \boldsymbol{Z})\ z : \boldsymbol{R}[x]$
$\quad \textbf{pre} \quad (n \neq 0) \wedge (n + l \geq 0) \wedge (n - l \geq 1)$
$\quad \textbf{post} \quad z = Q_{nl}(r)$

To obtain an implementation we apply the arguments relating to the choice of `Expression(Integer)` as the concrete type for $\boldsymbol{R}$ and $\boldsymbol{R}[x]$ as in Section 3.2.2. The values of `Rab` and `Fab` are unchanged and this type is able to represent all values of Q_{nl}.

The source for this implementation can be found in Appendix B.2. Unfortunately we find that this implementation is slower that the one from Section 3.2.2. This seems to be due to the way that **axiom** simplifies expressions—the first implementation generates expressions that the **axiom** `integrate` function can evaluate more quickly than those generated by this implementation.

3.3.2 Satisfaction

There are now two parts to the question of satisfaction: firstly does the implementation satisfy its specification and secondly does this implementation satisfy the implementation

of Section 3.2.2? The answers to both these questions produces different paths through the reification diagram of Figure 3.1 in Section 3.1. The proof that this implementation satisfies its specification is the same as that used for the first implementation: assuming the semantics of **axiom** are the same as the mathematics used in the specification all that is required is a correctness proof of the implementation of the double summation and the Kronecker Delta function.

The second question is slightly more complicated. Since the two programs will be used by calling `Fab` with different values of n_a and n_b, we can see this question is the same as asking whether `Fab'` satisfies `Fab` (for clarity the functions from the implementation of this section will be primed to distinguish them from their unprimed counterparts from the implementation of Section 3.2.2). Since these two functions are identical the satisfaction question is whether `Rab'` satisfies `Rab` *i.e.*

```
Rab'(na, la, nb, lb) ==
   integrate(Qnl(na,la) * Qnl(nb,lb) * exp(-alpha*r) * r**3,
             r=0..%plusInfinity)
```

must satisfy

```
Rab(na, la, nb, lb) ==
   integrate(Rnl(na,la) * r**3 * Rnl(nb,lb),
             r=0..%plusInfinity)
```

Both of these functions have been shown to satisfy their specifications and so we can use these specifications to obtain the verification condition

$$\int_0^\infty R_{n_a l_a}(r)\, r^3\, R_{n_b l_b}(r)\, dr \equiv \int_0^\infty Q_{n_a l_a}(r)\, Q_{n_b l_b}(r)\, e^{-\left(\frac{n_a+n_b}{n_a n_b}\right) r}\, r^3\, dr$$

which can be proved using the algebra of Section 3.1.2.

3.3.3 Further reification

Following the same pattern as in Sections 3.2.1 and 3.3.1 we reify to obtain a new specification using Equations 3.21–3.23:

$$
\begin{aligned}
r_{ab} &\triangleq \sum_{i=0}^{\beta} \frac{c_i\, i!}{\alpha^{i+1}} \\
c_i &\triangleq i^{th} \text{ coefficient of } Q_a(r)\; Q_b(r)\; r^3 \\
\alpha &\triangleq \frac{n_a + n_b}{n_a n_b} \\
\beta &\triangleq (n_a + n_b) - (l_a + l_b) + 1
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{Rab}\ (n_a : \boldsymbol{Z},\ l_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z},\ l_b : \boldsymbol{Z})\ z : \boldsymbol{R} \\
&\quad \textbf{pre} \quad (n_a \neq 0) \wedge (n_b \neq 0) \\
&\quad \textbf{post} \quad z = r_{ab}
\end{aligned}
$$

It can be seen from the specification above that an implementation of r_{ab} needs to be able to extract the coefficients of the polynomial $Q_a(r)\, Q_b(r)\, r^3$ and this places a restriction on the concrete data-types which can be used. In **axiom** only the polynomial and series types provide such an operation and so the `Expression(Integer)` type cannot be used to represent the real numbers here. Instead the floating-point type will be used.

Since floating-point numbers are an approximation to the real numbers, any implementation that uses them will not satisfy either its specification, or the implementation of Section 3.3.1. However, since the results can be computed to an arbitrary precision at the expense of increased computation time, the lack of satisfaction is not too significant for this final implementation. It also gives us an opportunity to see if there is a noticeable difference in the results of the different implementations. For simplicity the concrete data-type `UnivariatePolynomial(r,Float)` was chosen to represent $\boldsymbol{R}[x]$. The code for this implementation can be found in Appendix B.3.

3.3.4 More satisfaction

Using the methods described in Section 3.2.3 we find that this implementation does not satisfy its specification. This is because floating-point type only approximates the real numbers albeit to an arbitrary precision in systems such as **axiom**. Instead we may wish to weaken the satisfaction requirement and ask whether this implementation satisfies the implementation of Section 3.3.1 to within an acceptable degree of numerical error in the results. While this is almost certainly possible it is likely to be unpleasant and will not be

discussed here. Leaving aside the issues of approximating the real numbers with floats, if we wish to show that this implementation satisfies the previous one we need to consider the verification condition that the integral

$$\int_0^\infty Q_{n_a l_a}(r)\, Q_{n_b l_b}(r)\, e^{-\left(\frac{n_a+n_b}{n_a n_b}\right) r}\, r^3\, dr$$

can be implemented as a summation. This requires the use of the identity from Section 3.1.2

$$\int_0^\infty c_i x^n e^{-\alpha x} dx = \frac{c_i\, n!}{\alpha^{(n+1)}}$$

The proof that this can be implemented using a "`for`" loop is similar to the proof in Section 3.2.3 and need not be shown here.

3.3.5 Summary

In Section 3.3.1 we reified the specifications of Section 3.2 to obtain a more efficient implementation. Unfortunately the new implementation generated different (but algebraically equivalent) expressions than the first and was actually slower than it. In Section 3.3.3 the reification process was repeated and a third implementation was obtained. This was found to be significantly faster than the previous implementations and could easily be translated into "traditional" programming languages such as Fortran or C. However, we note that the latter implementation does not satisfy its specification since it uses the **axiom** floating-point data type which only approximate the real numbers. We pointed out that the satisfaction argument could be weakened to permit the use of this type provided that we allow a certain degree of numerical error in the results. Ignoring this aspect of data reification we pointed out that the implementation still satisfies the operation decomposition requirements.

3.4 An alternative direction

In Section 3.1.2 we stated that the integral for r_{ab} could be represented as the Laplace transform of the polynomial $Q_a(r)\, Q_b(r)\, r^3$. Since **axiom** provides a function for computing Laplace transforms we are able to embark upon a different reification path to that taken in the previous sections. In this section we reify Equations 3.10 and 3.11 to obtain two more

implementations. We do not, however, make any attempt to show that they satisfy their specifications or other implementations since the arguments are the same as those used in the previous sections. We begin by discussing issues of specification matching—following from our experience with this particular case study, such techniques could have enabled us to choose the Laplace transform path much earlier in our development.

3.4.1 Specification matching

When we first started this case study we were not aware of what a Laplace transform was, nor did we have any reason to expect that it would enable us to produce more efficient implementations than those created from the reification path of Sections 3.2 and 3.3. We only stumbled across this technique while browsing the **axiom** libraries looking for alternative integration routines after the case study had been completed. If the specification matching techniques of Wing [85] and Zaremski [88] been available to us then we may have discovered it sooner.

Specification matching is a way to compare two software components using their specifications. This enables functions or other software components to be identified and located using their behaviour rather than keywords in their names. For it to be successful one must be able to obtain both exact and inexact (or *relaxed*) matches under the control of the user. In the context of this case study we would have found it extremely valuable to have been given the option of searching for a function whose post-condition was that it returned the value of

$$\int_0^\infty f(x) \exp(-sx) dx$$

for some function $f(x)$. Of course writing down a specification of such a function may not be easy, especially when attempting to capture notions such as "f is a function of x".

3.4.2 Reification again

In this section we reify Equations 3.11 and 3.10, and Equation 3.16 to obtain two new specifications and implementations. Operation decomposition is simple algebraic manipulation and is similar to that used in Section 3.1.2. Likewise the data reification arguments are the same as those of Section 3.2 and so we do not provide any satisfaction proofs—they are

essentially the same as those given already.

The Laplace transform of a function f (which satisfies various conditions defined in standard texts such as [87, page 257]) with respect to x may be written

$$\mathcal{L}\{f(x)\} = \int_0^\infty f(x)\, e^{-sx}\, dx \tag{3.26}$$

where s is referred to as the transform variable. From Equation 3.11 it is clear that Equation 3.10 can be expressed in this form. Thus the specification of r_{ab} in Section 3.2 can reified to give

$$\begin{aligned} r_{ab} &\underline{\triangle} \quad \mathcal{L}\left\{T_{n_a l_a}(r)\, r^3\, T_{n_b l_b}(r)\right\} \\ s &\underline{\triangle} \quad \frac{n_a + n_b}{n_a\, n_b} \end{aligned}$$

$$\begin{aligned} &\texttt{Rab}\ (n_a : \boldsymbol{Z},\ l_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z},\ l_b : \boldsymbol{Z})\ z : \boldsymbol{R} \\ &\quad \textbf{pre} \quad (n_a \neq 0) \wedge (n_b \neq 0) \\ &\quad \textbf{post} \quad z = r_{ab} \end{aligned}$$

where $R_{nl} = \exp(-r/n)\, T_{nl}$ and where s is the transform variable. The specification of T_{nl} is

$$T_{nl}(r) \ \underline{\triangle}\ N_{nl} \left(\frac{2r}{n}\right)^l L_{n+l}^{2l+1}\left(\frac{2r}{n}\right)$$

$$\begin{aligned} &\texttt{Tnl}\ (n : \boldsymbol{Z},\ l : \boldsymbol{Z})\ z : \boldsymbol{R}[x] \\ &\quad \textbf{pre} \quad (n \neq 0) \\ &\quad \textbf{post} \quad z = T_{nl}(r) \end{aligned}$$

where N_{nl} is specified in Section 3.2. The choice of concrete data-types is the same as for the first implementation and the source code can be found in Section B.4.

The reification of the specifications given in Section 3.3 is even easier since Equation 3.16 is already expressed as a Laplace transform

$$
\begin{aligned}
r_{ab} &\triangleq \mathcal{L}\left\{Q_{n_a l_a}(r)\, Q_{n_b l_b}(r)\, r^3\right\} \\
s &\triangleq \frac{n_a + n_b}{n_a\, n_b} \\
&\texttt{Rab}\ (n_a : \boldsymbol{Z},\ l_a : \boldsymbol{Z},\ n_b : \boldsymbol{Z},\ l_b : \boldsymbol{Z})\ z : \boldsymbol{R} \\
&\quad \textbf{pre} \quad (n_a \neq 0) \wedge (n_b \neq 0) \\
&\quad \textbf{post} \quad z = r_{ab}
\end{aligned}
$$

Again the choice of concrete data-types is the same as for the second implementation and the specification of $Q_{nl}(r)$ can be found in Section 3.3. The source code for this implementation can be found in Section B.5.

3.4.3 Summary

In this section Equations 3.11 and 3.10 were reified to obtain a new specification which made use of the Laplace transform. Using the operation decomposition techniques of Section 3.1.2 and the data reification techniques of Section 3.2 we obtained a new implementation which was significantly faster. The resulting speed improvement is almost certainly due to the selection of the `laplace` operator by operation decomposition. As before we continued the reification process to obtain another implementation based on Equation 3.16. Unfortunately this is slower than the first version for the same reasons as given in Section 3.3. No satisfaction arguments were given since they are essentially the same as those found in previous sections.

In Section 3.4.1 we briefly looked at the subject of specification matching and its application to this case study. We concluded that if this technique had been available to us we may have been able to select the `laplace` function during operation decomposition much sooner in the design process and thus reduce the development time.

3.5 Summary and issues arising

In this chapter we have investigated the use of reification in the context of computer algebra systems as a way of producing successively more efficient implementations from an initial mathematical specification of the problem. We used the computation of the strengths of

emission and absorption lines for a hydrogen-like atomic oscillator as the problem to be solved, and in Section 3.1 we made use of operation decomposition in the form of simple algebraic transformations to enable us to write suitable VDM specifications. From these specifications, implementations were written to provide quantative comparisons on the benefit of reifying abstract specifications into more concrete ones. In Section 3.4 we discovered an alternative reification path which produced significantly faster implementations. We noted that the specification matching techniques of Wing [85] and Zaremski [88] might have been useful in this respect by allowing us to select the alternative routine much sooner in the development process.

The reification process that we used can be seen graphically in Figure 3.1 below. The main path is from **AIM** at the top to **OBJECTIVE** at the bottom but during the development several other paths were taken such as the implementation of Equation 3.16. Each of the implementations can be regarded as a reification of the the one above it in the diagram, hence the proofs that one implementation satisfied another. Since the level 3 implementation does not satisfy the level 2 implementation (see Section 3.3.3) due to the choice of concrete data-types, the reification path is marked by a curly arrow. Clearly one does not need to restrict reification to a linear chain of steps.

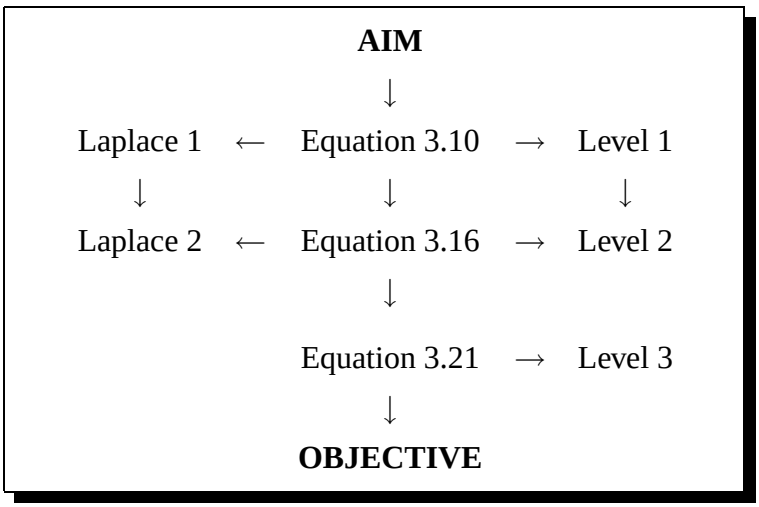


Figure 3.1: Reification Diagram of Case Study.

A number of issues arising from this case study are discussed next.

3.5.1 Implementing real numbers

Implementers of programs requiring numerical results often come across the problem of choosing a concrete data-type for a specification requiring the use of real numbers. For many the number of solutions is severely limited by the chosen implementation language. For example, C and Fortran can only support floating-point values of a finite precision unless special libraries are available and thus the programmer has little choice but to use them. Computer algebra languages such as that in **axiom** allow the implementer much more freedom. In this case study all of the programs except the most efficient were able to use the `Expression(Integer)` data-type which was able to represent the subset of real numbers that were required without any approximations. This type is a natural way of expressing real numbers since it is very close to the method used by mathematicians using pen and paper. The interface specifications and the justification of the data reification ensured that this was a valid choice.

The use of floating-point numbers as a concrete data-type for the reals is generally acceptable but a full specification ought to deal with the approximations and facilitate reasoning about the errors introduced. This is possible in theory but can be non-trivial (see, for example [39]). Work on constructing and reasoning about the real numbers is a current research topic [38] and here the power of computer algebra systems may be harnessed to aid theorem provers [40]; a formal development of the IEEE floating-point system is given in [4].

3.5.2 Reification of computer algebra programs

In [50] reification is considered in two ways. Firstly data reification is used to transform abstract data-types used in the specification into concrete data-types that can be easily implemented in the chosen programming language. Secondly operation decomposition is used to convert the operations relating to the abstract data-types into primitives available to the implementer. Working with a computer algebra system such as **axiom** means that the abstract data-types used in the specification are often available as concrete data-types of the language, particularly for small scale applications. Thus data reification can be used to select a particular data-type and to justify the choice.

With this case study there was little data reification involved except for the choice of a concrete data-type for implementing real numbers. However, the algebraic specification

was successively reified from an abstract (and inefficient) level to the concrete producing more efficient implementations at each stage. In fact most of the operator decomposition involved the standard algebraic manipulations that a physicist would normally perform when faced with similar equations even if they do not use this terminology.

3.5.3 Scaling up to larger programs

With larger applications the reification process is likely to be slow and probably very tedious. Data reification will probably play a bigger *rôle* than in this case study since the abstract data-types used in the specification are probably not available as concrete data-types in the implementation language at least at the top level. However, the use of formal methods and providing mathematical and/or logical justifications for any reification is likely to reduce the number of errors introduced into the software. It also helps the developer to gain insight into the problem before any code is actually written [78, page 124]. Once programming has started, developers are unlikely to want to return to the design stage unless there are severe problems. Instead a workaround would be sought which could lead to further problems in the future. Even with a full specification problems may be discovered, but these ought to be solved more safely with the knowledge of how much of the system will be affected by the change. The main obstacle facing anyone contemplating the formal development of software for computer algebra systems is the sheer size of the problem and machine assistance becomes increasingly more necessary. This subject is considered in more detail elsewhere in this thesis.

3.5.4 Other implementation languages

Implementations in general purpose programming languages such as Fortran and C are possible but require specially written libraries for representing and manipulating polynomials. The author has produced C and Fortran versions of the second implementation of Section 3.3 and the resulting code is around four times longer than the **axiom** versions. However, these programs have the advantage of being compiled directly into machine code and therefore execute extremely rapidly. An implementation written in Aldor would also be possible assuming suitable libraries were available.

3.5.5 Limitations of computer algebra systems

In Section 3.1.3 we found that although **axiom** can be used to develop programs quickly and easily, it does have limitations. As a result we were unable to utilise **axiom** to help with the more difficult parts of operation decomposition of Section 3.1. However, the implementations which were produced benefited significantly from the types and functions that **axiom** provides, such as the representation of polynomials and the integration functions. We also noted that while systems such as Mathematica were able to solve integrals which were outside the scope of **axiom** we were uncomfortable that unspecified assumptions were made on various parameters. Thus even though Mathematica was able to compute the correct answer to our problem we were left wondering if it would always make the same assumptions that we did.

Chapter 4

Design of Larch/Aldor

In this chapter we describe the syntax and semantics of the Larch/Aldor BISL which forms the central focus of our work. We begin in Section 4.1 by reviewing several existing Larch BISLs, their syntax and particular features, and use the results to guide our discussion of the requirements and design issues for such languages. In Section 4.2 we introduce the syntax and semantics of Larch/Aldor, and then in Section 4.3 we write LSL specifications of the Larch/Aldor store model based on the work of [12] for Larch/C. In contrast to their work and in common with that of other Larch BISLs, the specification of our store model is written entirely in LSL rather than as a translation from a specification language such as Z. We finish the chapter by reviewing issues which the design of Larch/Aldor has raised and suggesting future work.

An introduction to Aldor is given in Appendix A.

4.1 Introduction

Before we can design a new Larch BISL for Aldor and **axiom**, we must look first at existing Larch BISLs and investigate the issues which are involved, such as the structure of annotated programs. We begin with a review of some existing members of the Larch BISL family, highlighting their syntax and the features of the programming language they were designed to work with. Next we consider the issues which specifically relate to the design of Larch/Aldor, and the concept of a store model on which our logical theories are based.

4.1.1 A review of existing Larch BISLs

In this section we examine briefly the syntax of several prominent Larch languages and identify their main features. The results from this review will be used to mould the syntax of Larch/Aldor since we wish to remain consistent with existing Larch BISLs where possible. As described in Section 1.3.3, the main purpose of a Larch BISL is to provide clear and concise documentation for procedures and abstract data-type methods. At the time of writing there are 11 Larch BISLs of which Larch/CLU [84] was the first. The others are at various stages of development with perhaps Larch/C [36], Larch/Modula-3 [51] and Larch/C++ [59] being the most well-known.

Although the Larch system was designed with tool support in mind, few of the Larch BISLs actually have tools. Notable exceptions include the LcLint tool for Larch/C [24] and Penelope for Larch/Ada [35].

Larch/CLU

Larch/CLU [84] was the first of the Larch family of BISLs and formed the basis for the syntax and design of the others. CLU is a statically typed language with automatic garbage collection and the features that Larch/CLU caters for include clusters (abstract data-types), exception handling, iterators and type-parameterised (polymorphic) procedures. Larch/CLU specifications follow the syntactical structure of the CLU features which they are describing and are kept separate from the actual implementations. Larch/CLU uses the keywords `pre` and `post` to specify the pre- and post-conditions of a procedure and `mutates` to specify objects which *might* have their value modified by the procedure. Clusters may have interface specifications—the `uses` clause provides a link to LSL traits while the `provides` clause defines a mapping from CLU types to LSL sorts and can be used to indicate whether the ADT is mutable or not.

The `changes` clause is provided by Larch/CLU to indicate objects that may have their bindings altered by the procedure which can occur when *own* variables (similar to `static` variables in C) are used. The `remembers` clause can be used to list any *own* variables which are defined by the procedure. CLU also provides universally quantified parametric polymorphism and the Larch/CLU `where` clause may be used to impose restrictions on the set of possible types that a procedure accepts. This allows the specifier to insist that any

instantiation of a type parameter will provide certain operators.

Below is a fragment of a Larch/CLU specification of a stack abstract data-type showing some of the concepts described above. It was abstracted from [84, page 154]:

```
stack = cluster is empty, grow, read
   uses StackOfInt
   provides mutable stack from StkI
      empty = ...
      read = ...
      grow = proc(st:stack,i:int)
         pre     true
         mutates st
         post    st' = push(st^,i^)
         end
end stack
```

In addition to the clauses described above, Larch/CLU provides several built-in operators to assist specifiers. The `mutates` operator can be used to indicate which objects are actually modified by the procedure while the `new` operator specifies objects which are newly created and are not aliased by an existing objects. The `returns` operator indicates that the function terminates normally while `signals` is used to specify that an exception is raised (abnormal termination).

Larch/C

The syntax of Larch/C [36] specifications is similar to that of Larch/CLU although the keywords have been altered, perhaps to improve readability. One major difference is that Larch/C (LCL) specifications are stored in files separate from the C implementations rather than appearing in-line, or as separate declarations as they do in Larch/CLU. The motivation for this is that a typical C module often consists of a header file defining the module interface and a source file containing its implementation. A typical module written using Larch/C will use an LCL interface specification file in place of the standard C header file (which can be mechanically created from the LCL file).

Pre- and post-conditions are introduced by `requires` and `ensures` clauses and possible modifications to client visible state are declared using the `modifies` clause. Since C does not provide any special facilities for abstract data-types, the features used in CLU cluster

specifications are not available. Essentially Larch/C concentrates on issues of memory management and modifications to client-visible state. Tan [80] added the concept of claims (implications of the specification which implementations must satisfy) which could be used for test-case generation or as lemmas for program verification. The semantics of Larch/C were addressed formally by Chalin in [12] which helped to resolve some inconsistencies. Their work is used as the basis of the Larch/Aldor store model (see Section 4.3).

Larch/C++

The ideas developed for Larch/C have been extended to Larch/C++ [59] and several new operators have been added such as `assigned()` and `allocated()`. Larch/C++ supports both partial and total correctness interpretations (Larch BISL specifications normally assume total correctness), and provides support for type invariants which allow the specifier to make assertions about the way that values of a type may evolve during the lifetime of a program. It is also being used as a vehicle for research into the inheritance and refinement of interface specifications [58]. Originally Larch/C++ interface specifications were placed in separate files just as they are in LCL but, more recently, examples using new comment markers '`//@`' and slightly different syntax suggest that C++ code could be annotated directly (see for example [56]). Below is a Larch/C++ specification of a function which is intended to compute integer square roots:

```
unsigned int IntegerSquareRoot(int n) throw(NegativeRoot);
//@ behavior
//@ {
//@    requires n >= 0;
//@    ensures  returns
//@         /\  (result*result) <= n
//@         /\  ((result+1)*(result+1)) > n;
//@ also
//@    requires n < 0;
//@    ensures  throws(NegativeRoot);
//@    modifies nothing;
//@ }
```

Note that the omission of the `modifies` clause in the first part of the specification is equivalent to the statement `modifies nothing` found in the second part. The `returns` predicate in the first post-condition asserts that the function terminates normally.

Larch/Ada and Penelope

Penelope [35] is an environment for interactive program development and verification of programs written using a subset of sequential Ada with Larch/Ada as the specification language. The primary goal of Penelope is the incremental development of verified programs and Larch/Ada has several constructs to support this which are not found in other Larch. These include the use of assertions as cut-points which allow procedure bodies to be partitioned into smaller units, each having their own pre- and post-conditions, and the addition of loop invariants. Like Larch/CLU, declarations of functions and procedures are annotated with interface specifications rather than the implementations themselves. However, loop invariants and cut-points are placed within the body of the code along with commands to guide the proof attempts of Penelope.

Penelope uses a syntax-directed editor which automatically generates verification conditions from the program being written. The user has the option of inspecting and attempting to discharge these verification conditions at any time and may decide to add extra lemmas or commands as annotations to their program as a result. Each time the user makes a change to the program the editor will perform semantic checking and update the set of verification conditions accordingly. The verification condition generator is based on predicate transformers which generate pre-conditions for a program fragment given a post-condition. The pre-conditions created with this technique are not the weakest pre-conditions and depend on the annotations of the program fragment. In contrast to the other Larch BISLs, Larch/Ada assumes a partial correctness of specifications and makes no attempt to deal with termination. The syntax of Larch/Ada also differs significantly from other Larch BISLs as can be seen in the following example from [35]:

```
FUNCTION bin_search(a: intarray; m, n, x: integer)
   RETURN integer;
   --| WHERE
   --|    IN     sorted(a, m, n)
   --|    RETURN k such that k>=m and k<=n and a[k]=x;
   --|    RAISE  not_present <=> IN not present(x,a,m,n);
   --| END WHERE;
```

The `IN` clause is the entry condition (pre-condition) while the `RETURN` clause places restrictions on the value returned by the function. The `RAISE` clause allows the effects of abnormal termination to be specified.

Larch/Speckle

Speckle [82] is a compiler for a subset of CLU which makes use of specifications to perform optimisations which are not possible for traditional compilers. The Larch/Speckle language was specifically designed for the compiler and is therefore closely related to Larch/CLU. Speckle programs may contain procedures which have one or more special implementations in addition to a default implementation. The special implementations each have a guard predicate which specifies the conditions under which they may be used. The Speckle compiler uses its knowledge of the program state at the point of each procedure invocation to select the first specialised implementation whose guard is satisfied. If none of the guards can be shown to be true then the default implementation is used. Information about the program state is obtained from a control-flow graph which is constructed using the interface specifications of procedures. The pre-conditions of procedures are assumed to hold before they are called and the post-conditions are used to define the program state afterwards.

Larch/Modula-3

Like CLU, Modula-3 has automatic garbage collection and so Larch/Modula-3 [51] does not need to provide support for the specification of memory management facilities like Larch/C and Larch/C++ do. However, it does permit the specification of concurrent processes as well as dealing with exception handling and sub-typing with specification inheritance. Larch/Modula-3 also allows client visible state to be specified as read-only or write-only in a similar way to the interface specifications of VDM [50]. The semantics of the language have been defined in terms of a translation into LSL [51].

Larch/ML

Although ML is primarily a functional programming language, it *does* have concept of state and it is this that Larch/ML [85] aims to capture. Since ML has a formal semantics, reasoning about Larch/ML programs ought to be simpler than those of Larch/C for example. Larch/ML provides support for exceptions and the specification of higher-order functions. One unusual feature is that type inference must be applied to its specifications since the meaning of an identifier depends on the context in which it is used.

Other Larch BISLs

Other Larch languages include Larch/Smalltalk [14] which has been used to investigate the use of specification browsers while Larch/Corba [77] and GCIL [61] are concerned with concurrency issues. Larch/VHDL has been used to model the imperative aspects of integrated circuit design such as heat dissipation.

4.1.2 Requirements and design issues

In Section 4.1.1 we examined several existing Larch BISLs in detail with the aim of highlighting their syntax and the different programming language features they have been designed to work with. Using the results of this survey we present briefly some of the design decisions involved with creating a new Larch BISL. Later in Section 4.2 we will define the chosen syntax of Larch/Aldor and in Section 4.3 we construct a store model for it.

Syntax issues

The syntax of a Larch BISL is an important issue: users must be able to construct specifications without needing to continually refer to a manual. Thus the syntax needs to be natural and must not hinder the user. Ideally all Larch BISLs would share the same basic keywords but as we have seen in Section 4.1.1 this has not happened. However, the format of specifications is the same consisting of one or more clauses, each preceded by a keyword to define its meaning. Likewise there ought to be a common set of pre-defined operators or predicates such as `defined()` or `modified()`.

Related to this is the issue of whether new clauses ought to be defined to help mechanical checkers or the user. For example, the meaning of the LCL statement `trashes(x)` is that the value of '`x`' may not be referenced anymore. However, it may be desirable to define a `trashes` clause with similar semantics to the `modifies` clause which clearly indicates which objects *might be* trashed by a procedure. This technique is advocated in [12] to eliminate a flaw in LCL arising from using the law of the excluded middle. For example, if the predicate (`trashes(x)` $\vee \neg$`trashes(x)`) is passed to a system which automatically normalises all statements then it will reduce to *true* and will probably be discarded. The result is that the information that the identifier `x` *might be trashed* cannot

be represented in such a system. Using a `trashes` clause avoids this problem since the notion that an identifier *might be trashed* comes from the semantics of the clause.

Location of BISL specifications

What is the structure of annotated programs? In Larch/C and Larch/C++ BISL annotations are stored in separate files while in Larch/CLU they are appended to procedure and function declarations. Larch/Ada takes this a step further by allowing blocks of code to be annotated as well. The advantage of using separate files is that they can be distributed to users along with pre-compiled libraries of code. This provides both the syntax and the semantics of the routines without the user needing to see the implementation. With inline specifications it may be necessary to construct a tool to automatically extract procedure interface specifications for distribution as documentation.

The issue of the structure of annotated programs is closely related to the granularity of BISL specifications. For example, Larch/Ada allows fine-grained specification by allowing arbitrary blocks of code to be annotated but Larch/C does not and only provides course-grained annotations.

Higher-order statements

Annotations for a language such as Aldor where functions and types are first class values can pose problems when the BISL is based on first order logic. Although there are obviously situations where statements are unavoidably higher-order, they can be often avoided by interpreting specifications about functions and types as statements about an instance of them rather than about all possible functions or types. For example, given the following definition of the `twice` function:

```
twice(AType:Type, func: AnyType -> AType)(x:AType):AType ==
   func(func(x));
```

any statements about the parameter '`func`' may be interpreted as first order statements about a particular function whose value is currently unknown. This is exactly the same

mechanism that LSL uses for parameterised traits—the trait parameters are simply names for substitutions which are applied when the trait is specialised.

Store model

At first sight it would seem sufficient to simply define the syntax and semantics of specifications written using Larch/Aldor. However, as described in the previous sections, the meaning of Larch BISL specifications is provided by the underlying LSL traits: these traits represent the model of the store of the programming language being used and their high-level operations enable Larch BISL specifications to be written concisely. These high-level operations are defined in terms of operations which relate objects to the values associated with them at any moment during the execution of the program. We define the store model for Larch/Aldor later in Section 4.3.

Other issues

The programming language which the Larch BISL is aimed at plays a significant part in its design. Specific features such as concurrency, the provision of abstract data-types, higher-order functions and types as values will all play a part. The semantics of inheritance in the programming language (if appropriate) will dictate the way in which interface specifications may be inherited and care must be taken to ensure that inconsistencies do not arise as a result.

What are the semantics of BISL specifications—are they based on total or partial correctness? If total correctness is used then functions whose pre-conditions are satisfied *must* terminate and do so in a state where the post-condition holds. In contrast, partial correctness means that functions whose pre-conditions are satisfied *might* terminate and that if they do then it will be in a state where the post-condition holds. Clearly a total-correctness view precludes the annotation of functions that do not terminate (for example the `abort` function in C). On the other hand a partial-correctness view cannot express the fact that a function will always terminate, even when the pre-condition holds.

Some Larch BISLs use total correctness while others use partial correctness. However, Larch/C++ appears to be unique by allowing the specifier to determine correctness inter-

pretation of a `requires`/`ensures` block. By combining these it is possible to capture the conditions of termination more precisely—the total correctness interpretation can be used to identify the conditions under which the function will always terminate while the partial correctness interpretation used to identify when it it will never terminate.

4.2 Syntax and semantics of Larch/Aldor

In this section we define the syntax of Larch/Aldor which is used throughout this thesis. It is by no means a complete and final description and may be developed and extended in the future. However, we believe that it is sufficient to express many, if not all, the concepts found in existing Larch BISLs. We begin in Section 4.2.1 with an extended BNF definition of Larch/Aldor interface specifications for Aldor functions. This is augmented in Section 4.2.2 with a description of the extra features for writing specifications about loops. Then in Sections 4.2.3 and 4.2.4 we consider the format of specifications for domains and categories. In Section 4.2.5 we extend the description of function specifications to provide support for functions as parameters and finally in Section 4.2.6 we decide where all these specifications will "live" and how they will be associated with the Aldor source code that they are describing.

4.2.1 Functions

Perhaps the most important aspect of any programming language is the support for functions and/or procedures. As a result most of the Larch BISLs concentrate on providing interfaces to functions. We follow Larch/C++ [57] by allowing a specification to be split into one or more segments where the behaviour of a given segment is based upon its pre-condition. If the specification is required to be complete then all cases must be covered, *i.e.* the disjunction of all the pre-conditions must be a tautology. To prevent ambiguity, each pre-condition ought to be distinct from all the others unless the behaviour of each "overlapping" specification is identical under the conditions which they overlap.

Figure 4.1 shows the extended BNF (EBNF) definition of Larch/Aldor specifications of functions. An *italic* typeface is used to represent non-terminals while `teletype` is used for terminals (quote marks are sometimes used for clarity). The Kleene star "$*$" indicates

specification	::=	*uses-block plain-block also-block*
uses-block	::=	*uses-tag uses-clause* | ε
uses-clause	::=	LSL traits with renaming of Aldor domains
also-block	::=	*also-tag specification* | ε
plain-block	::=	*clause**
clause	::=	*list-clause* | *logic-clause*
list-clause	::=	*list-tag identifiers*
logic-clause	::=	*logical-tag expr*
expr	::=	*expr*$_a$ *expr*$_1$
expr$_1$	::=	*binary-op expr*$_a$ | ε
expr$_a$	::=	*expr*$_b$ *expr*$_2$
expr$_2$	::=	"`=>`" *expr*$_a$ | ε
expr$_b$	::=	*expr*$_c$ *expr*$_3$
expr$_3$	::=	"`\/`" *expr*$_b$ | ε
expr$_c$	::=	*expr*$_d$ *expr*$_4$
expr$_4$	::=	"`/\`" *expr*$_c$ | ε
expr$_d$	::=	*expr*$_e$ *expr*$_5$
expr$_5$	::=	"`=`" *expr*$_e$ | ε
expr$_e$	::=	"`~`" *expr*$_f$ | *expr*$_f$
expr$_f$	::=	*id* "`(`" *exprs* "`)`" | "`(`" *expr* "`)`" | *id* | *number*
exprs	::=	*expr expr-list* | ε
expr-list	::=	"`,`" *expr expr-list* | ε
identifiers	::=	`nothing` | *id id-list*
id-list	::=	"`,`" *id id-list* | ε
id	::=	"`0`" | "`1`" | `[a-z, A-Z] [a-z, A-Z, 0-9]`* `['ˆ]`$^+$
logic-tag	::=	`requires` | `ensures` | `invariant`
list-tag	::=	`modifies` | `trashes`
uses-tag	::=	`uses` | `semantics`
also-tag	::=	`and` | `also`
binary-op	::=	Any binary operator except: $\wedge$, $\vee$, $=$ and $\Rightarrow$
number	::=	Any number, usually an integer

Figure 4.1: EBNF Definition of Larch/Aldor Function Specifications

that the preceding term may occur zero or more times while a "+" indicates that the term can occur at most once. Terms in square brackets are used to denote a choice from the set of possible terms. For example, the term [a-z, 0-9] represents any single lower-case letter or any single digit. Appending a Kleene star to such a term represents zero or more occurrences of (possibly different) lower-case letters or digits. The term ε represents nothing or emptiness.

Intuitively, a Larch/Aldor interface specification consists of a sequence of clauses, each starting with a tag such as `requires` followed by a logical expression or a list of identifiers. Our EBNF definition is not sufficiently powerful to enable us to specify that each type of clause within a *plain-block* can be used at most once, *i.e.* we cannot have two `ensures` together. We could have defined an EBNF choice operator to deal with this but we can see little benefit.

The definition of logical expressions is such that the logical connectives bind less strongly than other binary operators. However, logical equality binds more strongly than conjunction and disjunction; negation and function application have the strongest bindings although the use of parenthesis may be used to alter this. In the following sections we define the semantics of the clauses which are shown in Figure 4.1.

Requires

The `requires` clause defines the pre-condition of the function being annotated. Identifiers appearing in this clause represent the values of program variables visible in the outermost scope level of the function body, in the state immediately before the function is invoked. All the other clauses in the specification block assume that the pre-condition is true. If it is not then the other clauses in the block may or may not define the behaviour of the function. Thus the `requires` clause acts as a guard to the other clauses.

Ensures

This is the post-condition which will hold in the state after the function terminates provided that the pre-condition holds before the function is invoked. In our definition here we do not assume the specification represents partial or total correctness.

Invariant

Invariant clauses have already been used in some Larch BISLs (such as Larch/C++), but Larch/Aldor is unique in allowing them to be associated with functions. The invariant specified by this clause is defined to be true at all times throughout the lifetime of the

objects to which it refers. Its purpose is to allow statements to be made about objects visible to the body of the function which may continue to exist after the function itself has terminated. Possible target objects include those which are created dynamically (`fluid`) and those which have been declared in an outer scope (free variables).

It is important to note that the pre-state of objects listed in this clause is the state that they were in before the function was invoked. Likewise the post-state is the state that they are in after the function terminates.

Modifies

The identifiers listed in this clause represent client visible state which *may* be modified by the function during its execution. A function is permitted to execute without modifying any of the identifiers specified but may not modify any others. Note that the current definition of this clause only allows objects to be referred to by simple identifiers. It does not provide support for array or record elements. This is to simplify the EBNF definitions.

Trashes

The objects referenced by the identifiers listed in this clause may not be used by the client once the function terminates. This may be due to the fact that the memory associated with the object has been released back to the operating system. It may also be used to specify that the object is being used as a private workspace by the function and the client must not refer to it any more. Again the current definition only permits simple identifiers (see the description of `modifies` above).

Uses

This clause allows BISL specifications to refer to LSL traits where renaming and parameterisation provides a link between Aldor domains and LSL sorts. For example in the following Larch/Aldor program:

```
++} uses SetTrait(Integer for E, Set(Integer) for C);
++} modifies nothing;
++} ensures result = (x \in s);
member?(x:Integer, s:Set(Integer)):Boolean == { ... }
```

the Aldor domains `Integer` and `Set(Integer)` are associated with the LSL sorts `E` and `C` respectively. These traits provide the semantics for LSL operators such as $\in$.

4.2.2 Loops

Since we wish to generate verification conditions from Aldor programs which make use of Larch BISL specifications we must provide mechanisms for annotating programs at a much finer level than just function definitions. The next stage is to allow annotation of loops—we argue that a loop can be treated like a black-box whose behaviour can be described in the same way as a function or procedure. A loop has pre- and post-conditions and during its execution it may modify client-visible state *i.e.* any object which is accessible in the scope where it is defined.

To assist with the checking of loop specifications we use two particular clauses, one old and one new. The `invariant` clause associated with function specifications allows the specifier to supply a loop invariant. This is a logical expression which will be satisfied throughout the execution of loop and we require that the evaluation of the loop test must not affect the validity of the loop invariant. Normally an invariant would only refer to objects in any state (as opposed to the pre- or post-state) but occasionally it may be useful to refer to values of objects before and after each iteration of the loop. To this end the pre-state of an object is defined to be the state prior to the execution of the loop body after the loop test; the post-state is the state after a single iteration and before the loop test.

The second clause (`measure`) is new and is intended to help with termination arguments. It defines an expression whose value belonging to a well-founded set (monotonically decreasing with each iteration of the loop having a minimum value when the loop terminates). Such expressions may enable the behaviour of the loop to be investigated using proof-by-induction. The type of the expression is commonly a natural number.

An example of these clauses is given below:

```
++} requires  b = 0
++} modifies  nothing;
++} ensures   b = a^
++} invariant (a^ + b^) = (a' + b')
++} measure   a
while (a > 0) repeat {
   b := b + 1;
   a := a - 1;
}
```

The grammar describing loop specifications is the same as that given for functions in Figure 4.1 with the following changes and additions:

clause	::=	*list-clause* | *logic-clause* | *value-clause*
value-clause	::=	*value-tag expr*
value-tag	::=	`measure`

We note in passing that the type of the expression appearing a *value-clause* may have any type. The measure expression has a type which has a total order on its values and which has a finite minimal value.

4.2.3 Categories

Categories (see Appendix A.1) are closely related to LSL traits since they define the interface of domains and define properties of domains. Unlike LSL traits, Aldor categories do not provide axioms to describe the behaviour of exports of domains. However, one may argue that categories are where specifications ought to be placed, particularly if a category is intended to have a particular meaning rather than simply acting as a template for packages. Indeed this approach is adopted by Extended ML (see Section 1.4.1) which allows axioms to be defined describing the behaviour of members of ML signatures.

Categories themselves will not have interface specifications associated with them directly. However, category exports can be annotated since in many ways this is the natural place for a specification of their intended behaviour. For consistency we require that the interface

specifications of domain exports satisfy the interface specifications of the category exports so it is important not make the category-level specifications too strong. Default operations may have interface specifications and are treated in just the same way as specifications at the domain level. The difference is that the default operations may be overwritten by a domain so will the associated interface specification. Essentially the Aldor domain and category inheritence mechanisms will apply to annotations just as they do with the implementations.

4.2.4 Domains

Many of the interesting domains in **axiom** and Aldor are implemented as functors, that is a function which implements a mapping from types to types, and so it is natural to allow them to be annotated with suitable BISL specifications. For the purposes of this work a domain which is not parameterised by a type will be considered as a functor with no arguments and can also be annotated. Since domains do not usually modify any client-visible state when they are created the `modifies` clause would appear to be redundant. However, we have decided to retain it since it may be useful to some users. The pre- and post-conditions of domains will almost certainly be used to make statements about the types over which a domain is parameterised. Although this would generally involve higher-order statements these can be avoided by treating type parameters as place-holders for a substitution by a particular domain instance.

As with loops, a domain interface specification can make use of the `invariant` clause. This is particularly useful since Aldor and **axiom** domains persist after the functor which created them has terminated. Invariants associated with domains may place restrictions on the values adopted by its internal state and on the values returned by its exports. The pre- and post-states of objects representing an internal state are those states immediately before and after the object is mutated.

4.2.5 Functions as parameters

The definition of function interface specifications given in Section 4.2.1 needs to be extended to deal with parameters which are themselves functions. This is achieved by noting that functions are constant values and therefore their pre- and post-states are the same.

Therefore we can define the behaviour of functions in terms of their would be defined normally. The definition of Figure 4.1 will be extended with a `where` clause:

clause	::=	*list-clause* | *logic-clause* | *where-clause*
where-clause	::=	*where-tag where-body*
where-tag	::=	`where`
where-body	::=	Function definition with *specification* for body

For example, consider the `twice` function again which applies another function twice in succession. We require that the function being applied is pure *i.e.* it does not modify any client visible state. This may be achieved as follows:

```
++} modifies nothing;
++} where
++}    func(x:AType):AType == {
++}       modifies nothing;
++}    }
twice(AType:Type, func: AType -> AType)(x:AType):AType ==
   func(func(x));
```

In this example we write that `twice` is a pure function and that its function argument `func` is also a pure function. Note that if a function parameter also has function parameters then we can use `where` in its specification and so forth.

4.2.6 Design issues

In this section we resolve some of the remaining design issues which were raised in Section 4.1.2 The subject of higher-order functions has already been discussed in Section 4.2.5 and the provision of special BISL operators such as `defined()` will be considered later in Section 4.3.5. Here we deal with the location of BISL specifications, their context, their granularity and their semantics.

Location

We have decided that Larch/Aldor specifications ought to be embedded in Aldor programs as annotations rather than kept separately as in Larch/C. Not only does this enforce the idea

of specifications-as-documentation but it ensures that there can be no confusion between which specification is associated with which Aldor function or domain. One argument for keeping annotations in a separate file as with Larch/C is that developers can distribute them along with the program header files to provide interface documentation without revealing details about the implementation. However, Aldor developers can annotate exports of categories and distribute these to achieve the same effect. Alternatively a tool could convert a fully annotated domain into an annotated category definition.

Larch/Aldor annotations will appear as special documentation comments *before* the statement which is being specified. There are times when it might be more aesthetically pleasing to place annotations immediately after the opening brace of a block of statements or immediately after the == operator, but this is not possible due to the syntax of Aldor.

Context

The context of any interface specification will usually be determined informally from its definition. A mechanical checker is expected to know the location of all the necessary LSL traits which provide the background theory to the specification. However, the specifier can explicitly define which traits are used through the `uses` clause described in Section 4.2.1.

Granularity

With the exception of Larch/Ada, Larch BISLs are course-grained since the smallest program unit that can be specified is a function. We permit arbitrary Aldor program statements to be annotated providing the user with fine-grained specification. At present we provide no support for cut-points or intermediate assertions but their effect can be achieved by annotating a statement such as an empty block.

Partial or total correctness?

Larch/Aldor specifications are total correctness statements. We have not provided a way of allowing partial correctness statements to be made and this is left for future work. We would expect to follow the syntax of Larch/C++ [59] in this matter.

4.3 Larch/Aldor store model

In this section we write down the LSL model for the Larch/Aldor store which will form the basis for Larch/Aldor BISL specifications. We begin in Section 4.3.1 by describing how the model is constructed and structured. We have used the LCL store model of Chalin [12] to guide us since it has many of the features that we require and was specifically designed with the idea of the Larch methodology in mind. Like [12] our aim is to construct a fairly general store model suitable for use with a variety of different imperative programming languages. This is tailored to Aldor through the specification of its basic domains. Then in Section 4.3.2 introduce a model of the unsorted store and object dependencies; this is followed in Section 4.3.3 with the specification of sorts and the sorted store. In Section 4.3.4 the model is completed by linking LSL sorts used in Larch/Aldor specifications to the sorts in our model. Finally in Section 4.3.5 we describe how the model may be used and write specifications of a few primitive Aldor domains by way of example. Specification of other domains can be found in the appendices.

4.3.1 Overview

A formal model of computation is needed to provide a logical basis for reasoning about Larch/Aldor specifications, for example by providing meaning to verification conditions. The model will identify and capture concepts such as object dependency, well-definedness and the imperative nature of the Aldor language. The model for a programming language that we follow here divides the program state into two parts.

The first part is referred to as the environment and represents bindings from identifiers and other *lvalues* used in the program, to objects (possibly overlapping regions of memory). In any given scope the environment mapping is fixed. The second part is the store which is modeled here: it allows the time-*dependent* nature of imperative programs to be considered in a time-*independent* manner, mapping objects to values in a given store or program state. The mutation of the value of an object produces a new store in which the values of the unchanged objects are the same as in the original store and the values of the mutated objects reflect the new program state.

Usually one would define the model of the store using a specification system which is based

on higher-order logic such as Z [72] or PVS [68]. This is because some statements, such as those which involve reasoning about types and sorts, are naturally higher-order. However, store models for other Larch BISLs such as LCL [80], LC++ [57] and LM3 [51] have used LSL even though this is based on first-order logic. In contrast to this [12] uses Z to specify their detailed model of the LCL store with a translation into LSL.

In spite of the drawbacks, the author has chosen to use LSL for the Larch/Aldor store model and have used [12] as a guide. The traits that we give in Section 4.3.2 and onwards are essentially divided in the same way as in [12]. At times we deviate from their structure since we have chosen not to use Knuth's Web system. We also define certain axioms differently, usually by adding or removing logical implication and using logical equivalence instead. As in [12] we do not attempt to specify the environment. Instead we assume that the mapping from identifiers to objects will be undertaken by whoever, or whatever, translates the Larch/Aldor specifications into LSL or LP.

Before modeling the Larch/Aldor store we need to consider how Aldor categories, domains and functions might be dealt with in such a formalisation. As mentioned at in previous chapters and in Appendix A, the main feature that distinguishes Aldor from other imperative programming languages is its two-level object model. Since the LCL store model in [12] was designed for a single-level object model we need to briefly consider how to incorporate Aldor categories into our model and how to deal with the fact that types and functions are first class values.

Categories

Although Aldor categories are values of type `Category` and can therefore be assigned to variables, their use in type-forming expressions, such as domain construction, is restricted: in Aldor type expressions can only contain identifiers which have a constant value throughout the scope in which the type occurs. Since all interesting category expressions are constants, we do not need to incorporate them into the store model directly, even though they appear to be present in the runtime store of an Aldor program. Instead, we use LSL traits which model the categories found in Larch/Aldor specifications and use them to extend the basic store model described later.

Domains

Aldor domains representing abstract data types initially appear to present the same kind of problems as categories but we are able to appeal to the type constancy of the language to help us again—Aldor domains can be modeled by an LSL trait which is used to extend the basic store model. Indeed this is the approach taken in [12] with LCL and other Larch BISL authors.

Functions

Functions are immutable but since we are using first-order logic there is little that we can usefully write about them in our model. It may prove useful in the future to define a simple trait for functions from type T_1 to T_2 with function application being the only operator. This model of functions is fairly primitive although currying may improve matters. In fact it may be possible (and worthwhile) to translate Larch/Aldor annotations into assertions based on an LSL model of functions but we do not pursue this here.

4.3.2 Unsorted store model

We begin by defining a model of an unsorted store: a store whose values have no sorts associated with them. Such values are of little use in themselves but this formalism allows low-level coercions, such as `pretend` in Aldor, to be modelled easily. This is useful because an unsorted value may correspond to several different sorted values. For example, Aldor domains have two different sorts for their values: one being the sort used for the internal representation and the other being the external view of the domain itself.

A store represents a single state of computation and associated with it is a finite set of objects whose unsorted values are defined with respect to the store. Objects represent *lvalues*, or (possibly overlapping) regions of memory: the environment provides a mapping between identifiers, record fields *etc*, and these objects. We do not deal with pointer values in this model: to do so we need to be able to map a pointer value in a specific store to an object (or its value).

```
UnsortedStore:trait
   includes   Set(Obj, Set[Obj])
   introduces
      empty   :                    → UStore
      mutate  : Obj, UStore, UVal → UStore

      value   : Obj, UStore       → UVal
      objects : UStore            → Set[Obj]
   asserts
      UStore generated by empty, mutate
      UStore partitioned by value, objects
      ∀ u,u':UStore, o, o1:Obj, v:UVal
         objects(empty)            == {};
         objects(mutate(o, u, v))  == insert(o, objects(u));
         value(o, mutate(o, u, v)) == v;
```

In [12] `value` is treated as a partial function defined only for objects which have values in a given store. In contrast we view it as an under-specified total function: we do not expect it to be applied unguarded to objects which do not have values in a given store.

Dependencies

In many imperative languages, the value of one object may depend on the value of other objects in the store. If the value of an object x can be changed by mutating the value of object x' then we say that x depends on x'. For example, the value of a record depends on the values of its fields and *vice versa*. It is also useful to be able to express the fact that a group of objects are mutually independent *i.e.* for any pair of objects in the group, neither object depends on the other.

The existence of object dependencies means that `mutate` may produce a new store in which the values associated with more than one object are different from those in the original store. Since we cannot specify this behaviour in our general model we are forced to under-specify `value`. However, for a store that models a particular program state, we hope that additional axioms would enable `value` to be completely specified.

```
Dependency:trait
   includes   Queue(Obj, Objects)
   introduces
      dependsOn   : Obj, Obj     → Bool
      independent : Objects      → Bool
      independent : Obj, Objects → Bool
   asserts
      ∀ s:Objects, o, o1:Obj
         dependsOn(o, o);

         independent(empty);
         independent(append(o, s)) ==
            independent(o, s) ∧ independent(s);

         independent(o, empty);
         independent(o, append(o1, s)) ==
            ¬dependsOn(o, o1) ∧ ¬dependsOn(o1, o)
            ∧ independent(o, s);
   implies
      Reflexive(dependsOn, Obj for T)
```

Our specification of object dependency is similar to that of [12]. The main difference is that we have defined the `independent` operators recursively which we believe helps to simplify any reasoning involving this trait. In particular proof-by-induction ought to be much easier since quantifiers do not appear in our definition.

4.3.3 Sorted store model

The unsorted view of the store allows the physical representation of the Larch/Aldor store to be modeled at an abstract level but it does not permit an interpretation of the values themselves. In this section we introduce the notion of sorts and associate a sort attribute with each object. Then we define our model of the sorted store and build on it with operations to simplify its use in other traits.

Sorts

A sort defines the interpretation of unsorted values and although each object has a single sort, the value of an object may be viewed as a value from other sorts. Furthermore, since

an unsorted value may not be a valid representation of a sorted value we need to define an operator to model this. We also introduce an operator which specifies the equality of two unsorted values viewed as values belonging to a particular sort:

```
Object:trait
   introduces
      sortOf : Obj ⟶ Sort

Sort:trait
   includes   Object, Set(Obj, Set[Obj])
   introduces
      objects  : Sort       ⟶ Set[Obj]
      validRep : Sort, UVal ⟶ Bool
      equal    : Sort, UVal, UVal ⟶ Bool
   asserts
      ∀ o,o1,o2:Obj, s:Sort, u,u1,u2:UVal
         (o1 = o2) ⇒ (sortOf(o1) = sortOf(o2));
         (o ∈ objects(s)) == (s = sortOf(o));

         validRep(s,u) ⇒ equal(s,u,u);
         (validRep(s,u1) ∧ validRep(s,u2)) ⇒
            (equal(s,u1,u2) = equal(s,u2,u1));
         (validRep(s,u) ∧ validRep(s,u1) ∧ validRep(s,u2)) ⇒
            ((equal(s,u,u1) ∧ equal(s,u1,u2)) ⇒ equal(s,u,u2));
```

The `objects` operator gives access to the set of all objects having a given sort while `validRep` is used to determine whether a given unsorted value is a valid representation of a value from the specified sort. Note that in [12] an additional equality operator is defined which allows unsorted values which have no valid sorted value to be compared. We have not included this in our model since we can find no use for it. This does not prevent the user from defining it later if it is found to be necessary.

Sorted store

Using the unsorted model and the specification of sorts from the previous section we can introduce the sorted model of the store. Following [12] we include a notion of "well-definedness"—an object is well-defined with respect to a given store if the value of the object is a valid representation of a value from that object's sort.

```
SortedStore:trait
   includes   UnsortedStore(Store for UStore), Sort, Dependency
   introduces
      wellDefined: Obj, Store → Bool
   asserts
      ∀ s:Store, o:Obj
         wellDefined(o, s) ==
            (o ∈ objects(s)) ∧ validRep(sortOf(o), value(o, s))
   implies
      converts wellDefined
```

Our approach to the specification of well-definedness differs from that of [12]. Rather than defining a function whose value is the set of all well-defined objects in a particular store we have chosen to provide a predicate which determines whether a particular object is well-defined with respect to a given store. Again we believe that this may help to simplify any reasoning involving this trait.

4.3.4 Sorted projection

The sorted model of the Larch/Aldor store given in the previous section is sufficient to define the meaning of any Larch/Aldor specification which contains only static object-dependency relationships. The `dependsOn` function from Section 4.3.2 allows aggregate data structures such as records, arrays and unions to be modeled while the `independent` functions can be used to capture the meaning of the `fresh` operator (`fresh(x)` states that the value of `x` is independent of the value of all other objects in a given store). To cope with the `modifies` clause we need to be able to identify all the objects which are associated with a given store (the pre- and post-states) and object dependencies.

However, we continue to follow [12] and specify a sorted projection of the store. This projection provides a view of a sorted store seen as though it only contains objects of a given LSL sort. The intention is that the projection provides a link between sorts used in the LSL formalisation of a Larch/Aldor specification and the underlying model of the store.

```
ProjectedOps(S):trait
   introduces
      raise:       Obj          → Obj[S]
      lower:       Obj[S]       → Obj
      sortName:    S            → Sort
      abstract:    UVal         → S

SortedProjection(S):trait
   includes   SortedStore, ProjectedOps(S)
   asserts
      ∀ v,v1:S, s:Sort, o,o1:Obj, ob, ob1:Obj[S], u,u1:UVal
         sortName(v) == sortName(v1);

         raise(lower(ob)) == ob;
         sortOf(o) = sortName(v) ⇒ lower(raise(o)) = o;

         lower(ob) = lower(ob1) == ob = ob1;
         (sortOf(o) = sortName(v)) ∧ (sortOf(o1) = sortName(v)) ⇒
            (raise(o) = raise(o1)) = (o = o1);

         (s = sortName(v)) ∧ validRep(s, u) ∧ validRep(s, u1) ⇒
             equal(s, u, u1) = (abstract(u) = abstract(u1));

         ∃ u (validRep(sortName(v), u) ∧ (v = abstract(u)));
```

In the specification above, `raise` and `lower` provide a way of converting between objects viewed from the sorted projection (members of sort `Obj[S]`) and objects viewed from the underlying store (members of sort `Obj`). In addition, `abstract` is a partial function which is used to model the conversion of unsorted values into values belonging to the sort being projected. The first axiom states that all values of the LSL sort being projected belong to the same sort in our model. Since LSL is based on first-order logic we cannot state in LSL the stronger axiom that distinct LSL sorts have distinct sorts in our model. Instead we state as part of our meta-theory that for any two LSL sorts S_1 and S_2

$$\forall\, v_1 : S_1,\ v_2 : S_2 \bullet (S_1 = S_2) \equiv (\texttt{sortName}(v_1) = \texttt{sortName}(v_2))$$

The second and third axioms show that `raise` is the inverse of `lower`. It is important to note that while the domain of `raise` is equal to the range of `lower` and *vice versa*, the domain of `raise` is a subset of all objects in our model due to the projection. Thus any application of `raise` needs to be guarded such as we do above. The next two axioms relate

equality between objects in the sorted store model with equality between objects viewed from the sorted projection. Note that we claim that `raise` and `lower` are bijective when the domain of `raise` is equal to the range of `lower`. This differs from [12] where they are injective and surjective. The penultimate axiom links the `equal` operator from the sorted store model to the LSL built-in equality operator for values in the projected LSL sort. The final axiom states that every LSL value has at least one unsorted value in our model.

Promoted operators

To simplify still further the interface between the LSL formalisation of Larch/Aldor specifications and our store model we promote three operators from the sorted store model to the sorted projection: these are `wellDefined`, `objects` and `value`. As one might expect each is defined in terms of its dual from the sorted store model and `lower`. In addition to the axioms for these operators we give some useful lemmas.

```
Promotions(S):trait
   includes SortedProjection(S), Set(Obj[S], Set[ObjS])
   introduces
      wellDefined: Obj[S], Store → Bool
      objects:     Store         → Set[ObjS]
      value:       Obj[S], Store → S
   asserts
      ∀ ob:Obj[S], s:Store
         (ob ∈ objects(s)) == (lower(ob) ∈ objects(s));

         wellDefined(ob, s)  == wellDefined(lower(ob), s);

         wellDefined(ob, s) ⇒
            value(ob, s) = abstract(value(lower(ob), s));
   implies
      ∀ o:Obj, s:Store, v:S
         sortOf(o) = sortName(v) ⇒
            (o ∈ objects(s)) = (raise(o) ∈ objects(s));

         sortOf(o) = sortName(v) ⇒
            wellDefined(o, s)  == wellDefined(raise(o), s);

         (sortOf(o) = sortName(v)) ∧ wellDefined(o, s) ⇒
            (abstract(value(o, s)) = value(raise(o), s));
```

The first axiom in the specification above states that if an object is a member of the set of objects associated with the projected store then the corresponding object is a member of the set associated with the underlying sorted store and *vice versa*. The second axiom completely defines the behaviour of the promoted `wellDefined` operator while the final axiom relates to the behaviour of `value`. The lemmas are mirrors of these axioms which arise because `raise` and `lower` are inverses. They are slightly complicated by the need to restrict the domain of `raise`.

Other properties

We finish our description of the store model by considering the set of objects associated with all the stores in a given theory and two related properties. The first property is that we have an infinite supply of objects which follows from the assumption that programs will not run out of storage. Since Aldor uses automatic garbage-collection this is not unreasonable and is certainly no worse than the same assumption made about C programs in [12]. The second property is that the objects associated with a particular store are a subset of all the objects associated with all stores. These properties are shown in the following trait:

```
OtherProperties(S):trait
   includes   Promotions
   introduces
      knownObjects: ⟶ Set[Obj]
   asserts
      ∀ o:Obj, s:Store
         ∃ o:Obj (o ∉ knownObjects);
         objects(s) ⊆ knownObjects;
```

One major drawback of this approach is that `knownObjects` is strongly tied to the concept of the sorted projection. We would prefer it to be independent but the trait hierarchy of the model presented here prevents this. Unfortunately changing the structure of our model may complicate its exposition and so we leave it as an exercise for the reader.

4.3.5 Using the model

In the previous sections we have defined a model for the Larch/Aldor store based on the Z and LSL specifications of the Larch/C store in [12]. The next stage is to extend the model with the aim of writing a trait which can be used to provide meaning for LSL and LP translations of Larch/Aldor interface specifications. We assume that such translations will be written in terms of the sorted projection and so we need to do two things. Firstly we need to specify operators such as `fresh` which can be used as useful shorthands in Larch/Aldor and secondly we need to specify Aldor domains. This latter task is large and there is neither time nor space to do this completely.

However, in [52] Kelsey has successfully constructed LSL specifications of the majority of the **axiom** category hierarchy pictured inside the cover of [48]. In doing so Kelsey has discovered some inconsistencies between the meaning of **axiom** categories and the mathematical objects that they are supposed to represent. More importantly for our work these specifications provide a large library of traits which we can draw upon. It is interesting to note that in [52] a top-down approach has been adopted starting with the most general **axiom** categories such as `SetCategory`. We have begun with a model of the Aldor store and must extend this upwards through the definition of basic types to provide support for this library.

Useful Larch/Aldor operators

In Section 4.1.1 we encountered operators such as `new` and `returns` which helped to improve the conciseness and clarity of BISL specifications. Here we define the meaning of operators available to Larch/Aldor specifiers.

In the trait below, the `fresh` operator is used to state that a given object is independent of all other objects in the specified store. This is useful for specifying the behaviour of program functions which construct new objects. Note that the freshness property does not indicate whether the value of the object is well-defined, or even if the object is defined at all in the store. Objects which have a value in a store are considered to be defined with respect to that store and this is captured by `defined`. Again note that this does not imply that their value is well-defined.

```
SpecialOperators:trait
   includes   Dependency, OtherProperties, UnsortedStore
   introduces
      fresh   : Obj, Store     → Bool
      defined : Obj, Store     → Bool
      trashed : Obj, Store     → Bool
   asserts
      ∀ o, o1:Obj, s:Store
         fresh(o, s)   == independent(o, objects(s));
         defined(o, s) == o ∈ objects(s);
```

The operator `trashes` in the specification above is notable for the lack of axioms defining its behaviour. It is intended to be used to state that the value associated with an object must not be referenced by the program in the specified store. This could be modeled by extending `UnsortedStore` to allow of object/value associations to be removed. However, this would require the `generated by` clause to be augmented which could result in more complicated induction proof attempts involving this core trait. We have chosen to under-specify it instead.

4.3.6 Issues

In the previous sections we constructed an LSL model of the Aldor store and gave examples of how it can be used to provide the background theory for Larch/Aldor specifications. We have followed the work of [12] where a Larch/C store model is defined in Z and then translated into LSL. In doing so we encountered a number of issues which we raise here.

Shortcomings

Two particular problems were noted during this work:

- From an aesthetic point of view it would be nice if the store model incorporated categories, domains and functions as values which could be placed in the store as they appear to be in Aldor. More practically the user of this model may find that the lack of support for domains too restrictive for the programs they are writing specifications

for but we do not anticipate this happening too often. However, we hope that our model does not preclude future extensions to cope with any such deficiencies.

- The lack of higher-order expressions in LSL was troublesome at times. However, it was really only missed at one point when we wished to state that distinct LSL sorts are distinct Larch/Aldor sorts in our model. This problem was resolved by adding it to the background theory and accept that one cannot reason with this axiom. One other place where the first-order nature of LSL may create difficulties is with the notion of finiteness *e.g.* when dealing with array objects.

In addition, we have not provided a formal model of the environment which maps *lvalues* such as identifiers and record fields, onto objects. Nor have we provided a way for pointer values in a given store to be mapped to an object associated with the store. Furthermore, our model of object dependencies is weaker than we would like because there is no concept that the value of one object might be contained in the value of another: we can only describe when the values of objects overlap in memory. This means that there is little that we can usefully say in our model about abstract data-types such as trees and other aggregates.

Soundness and completeness

We require our model to be sound, *i.e.* we cannot prove that something is true in our model when it is actually false. Without this property any formalizations based on our model would be meaningless since we would not have the "correct" model and so there would be little or no point in using it.

On the other hand we are not too concerned about whether our model is complete *i.e.* every logical statement using our model can be shown to be either true or false. Specifications are often not complete for a variety of reasons [60] and this model is no exception. For example, consider the `equal` operator in the `Sort` trait—it has only been defined for situations where the unsorted values are valid representations of a sorted value in the specified sort. We have not defined what happens if the unsorted values are invalid representations of a sorted value.

However, we have endeavoured to define a model in which as many operators as possible are completely defined so that any reasoning based on this model will not be unduly com-

plicated by under-specification. We feel that this model contains all the properties we want and might be regarded as informally complete in a tongue-in-cheek manner.

4.3.7 Conclusions

Even though Chalin [12] has done most of the hard work in laying down the framework for constructing a store model and then using that framework for LCL, producing a store model for Larch/Aldor has not been easy. Occasionally the author attempted to adopt a slightly different approach only to find that it was inappropriate. One area in which have have differed is in the way that we have specified the notion of mutually independent objects. While Chalin used relied on universally quantified statements about sets we have used a completely recursive definition without quantifiers. This will almost certainly simplify proof attempts and help to reduce the amount of user interaction required.

Another major difference between our model and that of Chalin is that we do not have a notion of *active objects*. We found that this created more problems and ambiguities than it resolved for a model of the Larch/Aldor store and so we changed the idea to using *known objects*. The environment for a program then provides a much cleaner and better method of determining the set of active objects—they are the objects which can be reached from the environment.

Finally the Larch/Aldor store model needed to take into account the Aldor concepts of categories, domains and functions as first class values. To a certain extent these issues have been side-stepped because we were forced to take them outside of the store model since we are using first-order logic. However, there is scope for them to be incorporated in the future should the need arise.

4.3.8 Future work

Our model has a lot of room for extension and improvement, in particular we have not described the environment which provides a mapping from *lvalues* such as program variables and record fields, into objects in the store model. We would also like to see the model of object dependencies extended to include the concept that one object “contains” another. This would make it easily to model and reason about trees and other aggregate data-structures.

Another area that has not been fully addressed is a model of functions and their application. This is also an area which does not appear to have been researched in the Larch community to date (Wing [85] and Leavens [57] touch on the issue but do not pursue it further). Perhaps this is because few of the existing Larch target languages allow functions to be used as first class values like Aldor does. Related to this is the need for studying domains as values, finding out what circumstances they are used in and what would be needed to include them in our model. Other topics include the use of dependent types, fluid variables and generators. More work needs to be done to to produce LSL traits for as many of the basic Aldor domains as possible. These can then build upwards towards the work of [52] where the majority of the **axiom** computer algebra libraries have been specified in LSL.

Looking further ahead the choice of which logical system to use to provide the semantics of BISL specifications is open to future development. Changing to higher-order logic is very appealing and may have a number of benefits. Possible candidates include HOL [34] and PVS [68] (the PVS specification language fits nicely with the Larch approach).

4.4 Conclusions

In this chapter we began in Section 4.1.1 by reviewing some of the existing Larch BISLs, looking at their syntax, semantics, granularity and the programming-language specific features that they were designed to support. Most of them are superficially similar in syntax and in the operators that they provide. Some of them store BISL specifications in separate files akin to C header files while others such as Larch/C++ [57] embed them in special comments. The latter approach appears to be the accepted method in recent times and we have adopted it for Larch/Aldor. Many of the Larch BISLs are coarse-grained where the smallest program unit that can be specified is the function. Larch/Ada [35] is one exception and allows cut-points and assertions to be associated with any program statements. This almost certainly follows from the fact that Larch/Ada was designed with program verification in mind and for this reason we have adopted a similar approach for Larch/Aldor. Unlike Larch/Ada however, Larch/Aldor does not provide support for guiding proofs of VCs generated by mechanical checkers. This is simply because we are not designing a program verification environment. Our goal is the generation of VCs and we do not involve ourselves with how they will be discharged (if indeed they are). Then in Section 4.1.2 we looked at the requirements and design issues associated with Larch BISLs, in particular the

subject of syntax, where BISL specifications ought to live (whether in-line or in separate files) and how higher order statements can be treated.

In Section 4.2 we began our description of Larch/Aldor by writing down EBNF definitions of the annotation language. We concentrate mainly on function annotations and then extend the description to cover annotations of loops and domains. We decided that categories did not need to be annotated *per se* but their exports (which are usually functions) may be. Support for functions with functions as parameters is provided by the `where` clause which allows the behaviour of function parameters to be specified in terms of their actual definition. Thus if $\mathcal{F}$ is a parameter which is a function then the `where` clause allows a definition of $\mathcal{F}$ to be provided with the function body replaced by its Larch/Aldor specification. We require that an instance of $\mathcal{F}$ satisfies its `where` specification. Finally we make several design decisions such as the location of Larch/Aldor specifications, their granularity and their semantics. We decided that Larch/Aldor specifications would be embedded inside special comments in Aldor programs and that any program statement can be annotated. A total correctness interpretation is used although support for partial correctness may be added in the future.

We finished this chapter with Section 4.3 which provides an LSL model of the Larch/Aldor store. We followed the approach taken by [12] for Larch/C although our specifications are written in LSL from scratch rather than as a translation from Z. Our model succeeds in avoiding the use of quantifiers except in two places. We believe that this helps to simplify any proof attempts which may be made using this model particularly involving traits such as `UnsortedStore` and `Dependency`.

The definition of the syntax of Larch/Aldor has not been completely formalised and there are a number of areas which need to be made more precise in the future. One example of this is the format of object identifiers—our description only allows simple identifiers to be used although in general we need to be able to refer to elements of arrays or records and function results. Similarly there are areas where the store model can be extended such as to provide support for functions, generators and fluid variables. Also needed is a model of the environment to provide a mapping from variables, array elements and record fields to objects.

Chapter 5

Lightweight VC Generation

5.1 Introduction

In Chapter 4 we defined the syntax and semantics of Larch/Aldor BISL annotations and in this chapter we introduce the idea of using lightweight program verification as a way of improving the robustness of programs written using Larch/Aldor. Annotated programs can be analysed, and from the information which has been deduced about the state of the program at each statement, verification conditions (VCs) may be generated. These VCs are logical statements which capture the validity of user-supplied pre-conditions, and if they can be shown to be true then the user will have increased confidence that their program will behave correctly according to its specification.

We begin in Section 5.2 by introducing the background to program verification: correctness, VCs and the different types of program analysis which may be employed. Then in Section 5.2.3 we describe our lightweight approach to program verification and in Section 5.2.5 we suggest possible uses of VCs. Next in Section 5.3 we outline the design decisions involved with the implementation of a lightweight VC generator, discuss the decisions that were taken and provide an overview of the way that our prototype works. Finally in Section 5.4 we review the status of our prototype and review related work.

5.2 Techniques

In this section we introduce several different techniques which can be used to generate verification conditions from programs. We begin in Section 5.2.1 by defining what it means for annotated programs to be correct and introduce the reader to verification conditions and inference rules for programs based on the ideas of Hoare [42]. Then in Section 5.2.2 we review the forwards and backwards approaches to program verification which are often expounded in the literature before moving on to introduce our approach of lightweight verification condition generation in Section 5.2.3. In Section 5.2.5 we highlight possible uses of VCs and finish with a brief look at different theorem proving systems which might be used to help the user discharge VCs that are generated from their programs.

5.2.1 Background

Here we define partial and total correctness, the notation that we will use in the rest of this chapter and then introduce inference rules for verification condition generation.

Partial and total correctness

The notation $\{P\}\,\mathtt{C}\,\{Q\}$ states that the program fragment `C` has the pre-condition P and post-condition Q; P and Q are the specification of `C`. Usually $\{P\}\,\mathtt{C}\,\{Q\}$ is interpreted as a "partial correctness" statement. This means that if `C` is executed in a state satisfying P and *if* it terminates then $\{P\}\,\mathtt{C}\,\{Q\}$ is true if Q is satisfied by the state after `C` has finished executing. If $\{P\}\,\mathtt{C}\,\{Q\}$ given a "total correctness" interpretation then it must be partially correct and if it is executed in a state satisfying P then `C` will terminate. Gordon [33] points out that there is not a standard notation for total correctness. To avoid confusion [33] uses $\{P\}\,\mathtt{C}\,\{Q\}$ for partial correctness and $[P]\,\mathtt{C}\,[Q]$ for total correctness. We will adopt this notation here.

Verification conditions and inference rules

The traditional approach which is taken to prove $\{P\}\,\mathtt{C}\,\{Q\}$ is to reduce the statement to a set of purely logical or mathematical formulae called "verification conditions" [33]. This

is achieved through the use of inference rules which allow the problem to be broken into smaller fragments. For example, the rule for assignment of the value of an expression e to the object v can be written

$$\frac{P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]}{\{P\}\ \mathtt{v} := \mathtt{e}\ \{Q\}}$$

Reading from the bottom, this states that to prove $\{P\}\ \mathtt{v} := \mathtt{e}\ \{Q\}$ we need to prove that $P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]$ where $Q\,[\mathtt{e}/\mathtt{v}]$ represents the formula Q with every occurrence of v replaced with e. Alternatively we can write

Assignment Rule
The verification condition from $\{P\}\ \mathtt{v} := \mathtt{e}\ \{Q\}$ **is** $P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]$

For example, the partial correctness proof of $\{x = 0\}\ \mathtt{x} := \mathtt{x} + \mathtt{1}\ \{x = 1\}$ generates the VC $(x = 0) \Rightarrow (x + 1) = 1$. As described earlier, the total correctness proof also requires that the assignment terminates *i.e.* the evaluation of e terminates.

The generation of VCs must be firmly based in a logical system and inference rules such as the one given above need to be carefully constructed for each type of action that a program make take. While our assignment rule is fairly simple, rules for loops and function or procedure calls are more difficult, particularly in the presence of operations which have side-effects. A nice introduction to program verification can be found in [62] while [30] provides a more detailed review of different techniques for a variety of programming methodologies including concurrency and non-determinism.

5.2.2 The traditional approach

As mentioned in the previous sections, the traditional approach to VC generation involves reducing a partial correctness statement about a program to a conjunction of logical formulæ which defines the correctness of the statement. In this section we consider two methods of VC generation, the first of which analyses programs from the start to the end while the other does the reverse.

Forwards analysis

The forwards analysis of a partial correctness statement is one which appears to be the most sensible at first. Given an annotated sequence $\{P\}\,\mathtt{C}_0;\,\mathtt{C}_1;\,\mathtt{C}_2;\,\ldots\,\mathtt{C}_n\,\{Q\}$ we use an inference rule which provides the semantics of $\mathtt{C}_0$ to obtain the post-condition Q_0 which holds after $\mathtt{C}_0$ has terminated. Note that Q_0 may be a conjunction of terms which includes P. This process is iterated to obtain the post-condition Q_n and the verification condition is $Q_n \Rightarrow Q$. It is clear from the simple example below that Q_n is the strongest post-condition which follows from P. It is also clear that Q_n can contain a large number of terms, many of which may be irrelevant. For example, given $\{true\}\,\mathtt{x} := 42; \mathtt{y} := 23; \{x > y\}$ we can apply forwards analysis as shown below in three steps reading from left to right:

pre	$\{true\}$	$\{true\}$	$\{true\}$
	`x := 42;`	`x := 42;`	`x := 42;`
		$\{x = 42\}$	$\{x = 42\}$
	`y := 23;`	`y := 23;`	`y := 23;`
			$\{x = 42 \wedge y = 23\}$
post	$\{x > y\}$	$\{x > y\}$	$\{x > y\}$

From this we obtain the VC $(x = 42) \wedge (y = 23) \Rightarrow (x > y)$ which is trivial.

A major benefit of this approach is that Q_n encapsulates all the information about the program fragment which can be derived from the partial correctness statement using a given set of inference rules. Thus Q_n represents a symbolic trace of the execution of the program fragment. It may be argued that Q_n could become too large to store or manipulate, but with current hardware and software technology systems of thousands of equations can be handled with relative ease. Furthermore, each term in Q_n is likely to be simple and discharging VCs may be possible by a very naïve normalisation process.

Backwards analysis

The backwards or goal-directed analysis of programs is more commonly used—the post-condition Q of the correctness statement is pushed backwards to obtain intermediate pre-conditions P_i and eventually the weakest pre-condition P_0. The VC is that $P \Rightarrow P_0$.

Applying backwards analysis to the program fragment from the previous example gives:

pre	$\{true\}$	$\{true\}$	$\{true\}$
			$\{42 > 23\}$
	`x := 42;`	`x := 42;`	`x := 42;`
		$\{x > 23\}$	$\{x > 23\}$
	`y := 23;`	`y := 23;`	`y := 23;`
post	$\{x > y\}$	$\{x > y\}$	$\{x > y\}$

This time we obtain the VC $true \Rightarrow (42 > 23)$ which is equivalent to that obtained from by forwards analysis. Note that inconsistencies encountered during the analysis will result in the weakest pre-condition *false* which means that the correctness statement is false. Unlike the forwards analysis described in the previous section we do not obtain much information about the program itself and thus require less memory or storage to represent it. However, this is a drawback if one requires something more than a simple "this VC implies the correctness of the program".

5.2.3 The lightweight approach

Our approach to verification condition generation is different from those described in the previous section. The assignment rule which was given earlier is relatively simple but the construction of rules for other features of a programming language such as Aldor is not so easy. In particular, determining the VCs resulting from calling a procedure which mutates its arguments is difficult. Instead we propose the use of lightweight verification condition generation where user annotations (in particular those associated with procedures and functions) are used to provide the operational semantics of program fragments instead of inference rules based on the semantics of the programming language.

Using our previous notation, $\{P\}\,\mathtt{C}\,\{Q\}$ might represent the correctness of a standard library procedure `C`. In any context which executes `C` we have the verification condition that P is satisfied in this context. From this we can assume that Q is satisfied in the new context after `C` has terminated and we use Q to define the new context. Forwards analysis of the program is used to generate VCs from each pre-condition and new contexts from each post-condition (it is unclear how backwards analysis can be applied here). This technique can be applied recursively so that each program fragment `C` can be checked against its annotation

$\{P\}\,\{Q\}$. Note that in practice the VC generator may provide additional annotations which can be derived from the semantics of the programming language so that it can increase the number of useful VCs which can be generated from a given program.

Our justification for this approach comes from the area of computer algebra systems which has motivated our work. We believe that it is more likely that programming errors will be due incorrect application of functions or procedures than due to mistakes in the the implementation of computer algebra routines. After all, many of the algorithms upon which these systems are based have been well studied, sometimes for centuries.

As a simple example, consider a function to compute the integer square root:

```
++} requires ~(x < 0);
++} ensures  (r*r <= x) /\ (x < (r+1)*(r+1));
++} modifies nothing;
iqsrt: (x:Integer) -> (r:Integer);
```

We trust that the implementation of this function satisfies its specification (refer to Section 4.2 for an explanation of the syntax and semantics), namely that

$$\forall x \bullet x \geq 0 \Rightarrow \texttt{isqrt}(x) = \lfloor \sqrt{x} \rfloor$$

From a statement such as "`a := isqrt(z)`" we generate the VC that $\neg(z < 0)$ holds beforehand, and infer that $(a * a \leq z) \wedge (z < (a+1) * (a+1))$ holds afterwards.

5.2.4 Multiple execution paths

The presence of multiple execution paths within the program fragment of a correctness statement raises problems particularly when forwards analysis is used. For example, while using forwards analysis we find that after the statement:

```
if (a < b) then
   min := a;
else
   min := b;
```

there are two possible values of Q_n, namely $(a < b) \Rightarrow (min = a)$ and $\neg (a < b) \Rightarrow (min = b)$. Should these two paths merged by conjoining the two Q_n predicates or should the analyser keep the execution paths distinct risking a potential combinatorial explosion in the number of program paths being maintained? If they are merged then the context used for each VC may be very complicated involving several case splits whereas keeping each path helps to keep the context as simple as possible at the expense of extra storage requirements. It may be possible to alleviate the potential combinatorial explosion by merging paths which correspond to the same program state as well as through judicious use of annotations by the user. For example if a block of program statements is annotated then multiple paths within that block can be treated as a single path outside the block.

5.2.5 Using verification conditions

Once we have generated a set of verification conditions from an annotated program, what can we do with them? Ideally we would proceed to prove that they were true but in practice this may be infeasible. For example, the GAP4 CAS [31] contains a small module which would generate an apparently simple verification condition. However, the proof of this VC relies on the "Odd Order Theorem" whose proof occupied an entire 255 page issue of the Pacific Journal of Mathematics [26]. Other examples might include statements about continuity of mathematical functions or computational geometry.

Generating verification conditions by hand is tedious even for tiny programs so we hope that they will be mechanically generated. Indeed later in Section 5.3 we describe our prototype to do just that. Once the verification conditions have been generated then there are various possibilities of what can happen next:

- An automated theorem prover or proof assistant could be used to help with proof attempts of VCs. For example, mechanised proof tools could be used to quickly eliminate trivial VCs leaving the user to concentrate on the remainder. These tools are unlikely to be able to discharge all valid VCs automatically but they may be able to assist the user with tedious tasks such as simplification or proof-by-cases.
- Obvious mistakes may be detected by a user more quickly than a machine. The validity or otherwise of VCs may be easily decidable by inspection. For example the statement $(\tan x)$ *is-continuous-on* $(0, \pi)$ is clearly false and this can be easily seen

by the use of the graph of $\tan x$ on the interval $(0, \pi)$. On the other hand, attempting to prove this mechanically with a theorem prover is hard, requiring a model of the real numbers which is a topic of active research [40, 47].

- Trivial VCs such as $(x = a) \Rightarrow (x = a)$ could be automatically discharged by the VC generator. Other VCs which are easily normalised to true using axioms from a theory associated with the problem could also be dealt with in this way. The benefit is that the user will not be inundated with numerous trivial VCs which make it difficult to spot the interesting or important ones.

- The user may appeal to their specialist knowledge or use authoritative sources. For example, the Odd Order Theorem mentioned above essentially states that a group which has odd order is soluble. If this was generated as a VC then an expert could accept that it is true and avoid a long and tedious proof from first principles.

- VCs may highlight additional constraints on the program that were omitted from its specification. Rather than attempting to prove such VCs the user may decide to augment the annotations appropriately. Even if they do not represent omissions the user may still decide to extend the annotations and alter the original specification.

- Finally the user may simply decide to accept some VCs on trust especially if there are other VCs which are considered to be more important or which merit further investigation.

Proof attempts which fail to show whether a VC is valid or invalid may indicate omissions from program annotations or from the background theory. In general though the VC might simply be too difficult to prove and the additional knowledge required to prove might be unavailable. VCs which are found to be invalid imply that there is a mistake, probably in the program or annotations but possibly in the theory used during the proof. If all VCs can be proved true then the user has increased confidence that their code behaves as expected in situations where the pre-conditions are satisfied. However, since this is relative to the specification it does not preclude the possibility of the program behaving in unexpected ways if there is a mistake in the specifications.

5.3 A prototype lightweight VC generator

In this section we describe the design and implementation of a prototype VC generator for use with Larch/Aldor programs. We begin in Section 5.3.1 by introducing the design decisions which affect the development of such a tool. Then in Section 5.3.2 we report on the current status of this tool, a lightweight VC generator which has been written in Aldor itself. This is followed by Section 5.3.3 which describes changes that were made to the Aldor compiler to support Larch/Aldor annotations and the way that the tool actually works. Finally in Section 5.3.4 we look at the lessons learnt from the implementation of the tool and then in Section 5.3.5 we review the decisions that we made during the design and implementation, and suggest how the tool could be developed in the future.

5.3.1 Design decisions

As with any software engineering problem, there are a number of design decisions which need to be made. Some of these are ostensibly simple but they may have far reaching consequences for the usefulness of the VCs which are returned by our tool.

Architecture

Figure 5.1 shows the outline of the architecture that we would like our VC generator to have. We envisage that a compiler for the target language would accept annotated programs as input and generate object files such as executables in the usual way. In addition to any warning and error messages that might be produced, the augmented compiler could automatically generate and display VCs. The compiler may also need to accept specifications which define the meaning of the program annotations as additional input. In the Larch world this would be in the form of LSL traits.

This architecture is simple but since it is monolithic, its behaviour cannot be modified by third parties. An alternative would be to allow the compiler to generator a representation of the input program complete with annotations for types, specifications and source code references (*e.g.* line numbers). This external representation could be analysed by external tools which are able to rely on the fact that the program is legal according to the compiler.

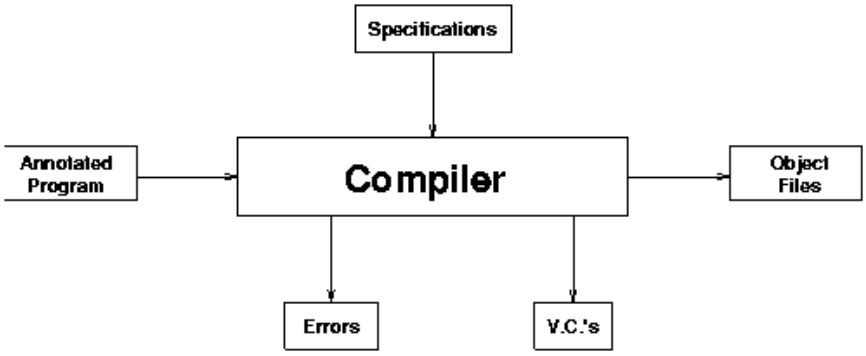


Figure 5.1: Ideal VC Generator Architecture

Simplification of VCs

Should the VC generator automatically perform normalisation or simplification of VCs before they are presented to the user? For example, given a complex VC that can easily be reduced to *true*, should the VC generator emit the original VC or nothing at all because the VC was trivial? On one hand the original VC may contain terms which could provide the user with useful information. However, too many trivial VCs may hide useful properties amongst a lot of irrelevant ones. This is really a task for artificial intelligence but the VC generator could provide ways for the user to control simplification to a certain degree.

Multiple paths

We mentioned in Section 5.2.4 that branch points and multiple paths create problems for program analysers such as this. Should paths be merged or kept separate?

Object language of VCs

What output or object language should be used for VCs? If the VCs are going to be processed by another tool such as an automated theorem prover then they need to be generated in a format which can be understood by that tool. This means that VCs must either be produced in the object language of the tool or in a neutral format which can easily be translated into the object language. One option would be to support more than one object language,

possibly including a neutral format such as LISP expressions.

Level of analysis

Should the analyser only process statements which occur at a specified level and use annotations to determine the behaviour of loops and other block structures? Alternatively can the analyser recurse into loop bodies and analyse them as separate units with respect to their annotations? How deep ought this recursion be and ought the analysis be performed breadth-first or depth-first?

Types within annotations

The issue of types within annotations is important. Ideally the analyser ought to be able to infer the types of most symbols, either from their context or from the type of the program identifiers which they represent. However, type inference and type checking are by no means small tasks and so the analyser may force users to annotate all identifiers with explicit type references.

Internal representation

Finally there is the issue of the internal data structures used by the analyser. In Section 5.3.2 we describe the use of a tree-based representation which mirrors the lexical structure of the program being analysed. This has the benefit that it is easy to navigate and lends itself to a simple recursive implementation. In addition there is a direct link between a node in the tree and a line in the program source which can be used to provide a context for the user. However, the lexical structure of a program does not completely correspond to the flow of control during execution and it can present difficulties. A better alternative is to use a control-flow graph similar to those used in compilers for performing optimisations. Changes in program state always take place along an edge of the graph which are labeled with formulæ describing its semantics. Nodes in the graph correspond to program constructs such as assignment, procedure call or loop (see the work of Vandevoorde [82] for example).

5.3.2 Current status

As we have already mentioned, a prototype VC generator for Larch/Aldor has been implemented in Aldor. This tool is a two-pass analyser which is able to accept simple annotated programs and generates files in the LP object language. The output files comprise of assertions which describe the state of the program after the execution of each statement. These statements are interspersed with VCs in the form of LP "`prove`" commands. In this section we describe the design decisions that were made that led to this implementation.

Architecture

The architecture that we have adopted for our prototype analyser (see Figure 5.2) is noticeably different to the "ideal" architecture of Figure 5.1. An existing Aldor compiler has been modified so that it can produce an annotated parse tree as well as executables depending on the setting of command line switches. The lightweight VC generator has been implemented as a separate tool that accepts the parse tree from the compiler instead of as an extra compiler phase.

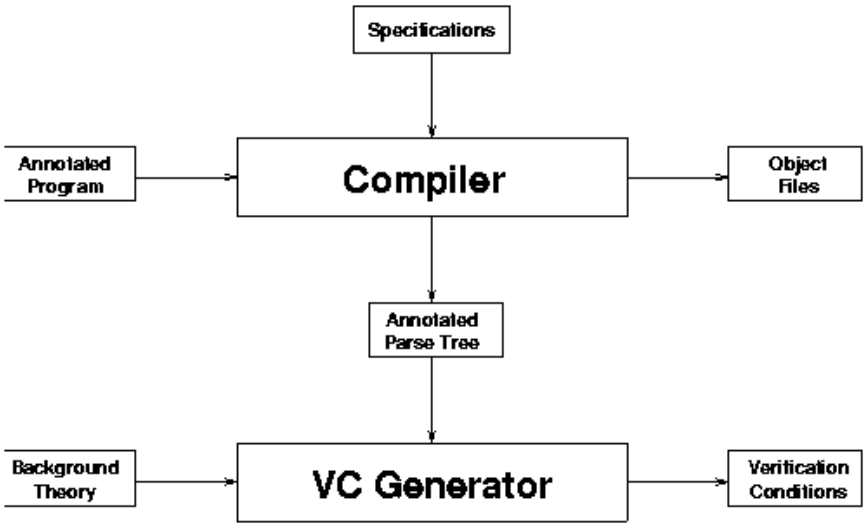


Figure 5.2: Prototype VC Generator Architecture

This approach was chosen for pragmatic reasons—on the fastest machine that we had at our disposal (a Sun SparcStation-10), it took between 4 and 8 hours for a complete rebuild of

version 1.1.7 of the Aldor compiler, libraries and test suite. The compiler distribution was well structured which meant that small changes to the source code only took a few minutes to rebuild the compiler executable. However, we decided that significant changes to the compiler would take too long to implement properly given the compilation time. Since there were existing facilities for generating parse trees in LISP format we decided to use them so that the compiler could produce similar output where each symbol was annotated with its type and user-supplied specification. This would require the least amount of change to the original compiler.

We decided that it would be better to make a number of relatively small changes to the compiler so that an annotated parse tree could be generated using command line switches. This output represents a syntactically and type-correct program and can be easily read in by external tools. It is important to us that the external representation requires little or no type inference since this is where a compiler can often spend much of its time.

Simplification of VCs

Our prototype does not perform any simplification or normalisation of VCs or statements about the program state. Again this provides the user with as much information as possible and does not make any assumptions about what normal form ought to be used or which formulæ are "simpler" than others. Also any simplification that our tool might undertake can be performed more efficiently by a proof assistant or automated theorem prover and with a greater degree of user-control.

Multiple paths

The original version of the VC generator kept multiple execution paths distinct and a tool was used to split the output into separate files where each file corresponded to a single execution path. Although this approach can suffer from combinatorial explosion we expected that the user would be aware of the problem and take appropriate action. However, the latest version of the VC generator adopts a different approach: assertions about the program state are guarded (using logical implication) by the condition under which a particular path may be followed. For an `if-then-else` statement, the guard is the conditional test for statements executed for the `then` branch and its negation for the other statements. Distinct

paths are merged using conjunction.

Object language of VCs

The object language of the LP proof assistant was chosen to represent VCs, primarily because LP was available with the Larch distribution and because of local expertise. Statements about the program state appear as assertions while VCs are given as `prove` commands. Due to the modular nature of our implementation we could use a different output format simply by producing a different "output" object.

Level of analysis

At the moment our prototype only examines the statements appearing at the top level of a program. If any statement has been annotated by the user with at least a post-condition then the behaviour of that statement is obtained from the annotation. Otherwise the analyser will recurse into the statement and analyse it just like any other statement. We would like the user to be able to control the level of analysis directly but this not an easy problem to solve. For example, how does the user refer to a particular program statement—line numbers are subject to change and are not always easy to deduce.

Types within annotations

We have already noted that type checking and type inference is a significant task especially for a language with the expressiveness of Aldor. As a result of this it was decided that our prototype analyser would not perform type inference or type checking. Instead all identifiers in user annotations must be given an explicit type which is taken on trust by the analyser. This reduces the readability of annotations but we feel that this was a price worth paying for a prototype.

Internal representation

The internal representation is a parse tree essentially the same as that generated by the augmented Aldor compiler. Although this representation creates difficulties when analysing

loops the analysis of other Aldor constructs is easy and naturally recursive. During the development of the prototype it was relatively straightforward to compare the internal data-structure of the VC generator to the Aldor program it represented.

5.3.3 Implementation details

In this section we describe the implementation details of our lightweight VC generator in more detail. We begin by briefly describing the changes that were made to the Aldor compiler and then outline the two passes which the analyser makes over its input.

Compiler modifications

As mentioned earlier, the Aldor compiler already had facilities to produce the parse tree in LISP format (as an AX file) but this format was not completely sufficient for our needs. The most important factor was that the AX files did not contain any additional type information above and beyond that which was present in the input source. Whereas the compiler could perform type inference to extract these types using built-in algorithms, we could not. Thus it made sense to allow the compiler to generate an annotated parse tree after type inference, where each identifier is decorated with its type declaration whenever it appears. This means that the Larch/Aldor specification of a function is immediately available whenever the function is applied and eliminates the need to examine libraries and other definitions to obtain this information.

The decision to generate annotated parse trees allowed us to reduce the number of changes which were needed to be made to the compiler to enable it to provide support for external program analysis tools such as VC generators. In addition to the generation of the annotated parse tree, we needed to extend the lexical and syntax analysers so that they could recognise Larch/Aldor annotations and store them internally. This was achieved by defining a specification node for the parse tree which associated an annotation with a program fragment. This was based on the existing nodes for recording documentation comments. Other changes to the compiler were needed so that the new parse tree could be recognised by the type inference, optimisation and object-generation phases. Essentially these were minor alterations so that, for example, the type of a specification node was the same as the type of the node being specified.

Originally we wanted Larch/Aldor annotations to appear immediately before the right-hand side of any definitions (*e.g.* `A` == *annotation* `B`) and to appear immediately after any declarations (*e.g.* in categories). However, while the latter was possible we discovered that the former was not due to the grammar of Aldor. As a result of this, all annotations must precede the declaration or definition of the identifier that they are referring to (see for example, Section 5.2.3).

Algorithms

The high-level view of our VC generator is one of a two-pass analyser. In the first pass we traverse the parse tree obtained from the compiler and pre-process all annotations while in the second pass the VCs are generated. The pre-processing phase involves cleaning up user-annotations and then renaming identifiers which appear in them so that their program state is made explicit. For example, given the program fragment:

```
++} requires a > 0;
++} ensures  a' = a^ + 2;
++} modifies a;
a := a + 2;
```

we note that the identifier '`a`' in the pre-condition is strictly a reference to '`a`' in its pre-state, *i.e.* '`a^`'. The **modifies** clause tells us that at most the variable '`a`' will be modified and so we create two logical symbols—'`a_0`' and '`a_1`' where the first represents '`a`' in its pre-state and the latter represents '`a`' in its post-state. After renaming we have:

```
++} requires a_0 > 0;
++} ensures  a_1 = a_0 + 2;
++} modifies a;
a := a + 2;
```

The pre-processing phase is recursive and as the recursion unwinds it is possible to check for violations of the `modifies` clause. If an identifier has the pre-state suffix i then after the pre-processing of the statement being annotated has completed this identifier must still be in state i unless it appears in the `modifies` clause. To improve the results that the VC generator can produce the pre-processing phase also creates internal annotations for

program statements based on the semantics of Aldor. An example is that an assignment may be given a post-condition which states that the logical symbol representing the post-state of the object being assigned to is equal to the symbol representing the value of the expression being assigned and that the identifier has been modified.

Most of the pre-processing is concerned with simple yet fiddly housekeeping tasks: these included tracking the state of Aldor symbols, and storing the names and types of logical symbols in the parse tree at the point where they are first used. The annotation of if-then-else and loop statements is quite involved since we need to be able to compare the state of all objects at the start and end of each execution path, and make assertions about them. Another source of considerable work was ensuring that each node in the program was assigned the correct type since the information obtained from the compiler only provided type information about identifiers. This could have been resolved by extending the compiler output but it was decided that a very simple type inferencing system using dynamic variables would be sufficient. This only required one or two extra lines of code per function.

After pre-processing, the generation of VCs is fairly simple since the majority of the hard work has already been performed. For each statement of the program the pre-conditions are analysed to create VCs based on the knowledge of the current program state. If a statement has not been annotated by the user with a post-condition then the analyser recurses to generate VCs for the statement before proceeding with the next one. Finally the post-condition is analysed to determine the new program state.

As a slightly longer example consider the following program fragment:

```
if (x <= y) then
   max := y;
else
   max := x;
```

If we assume that the symbols representing the identifiers ‘a’, ‘b’ and ‘max’ are initially in state 0 then the pre-processing phase will produce the following annotated program:

```
++} requires true;
++} ensures  (result_0  = (x_0 < y_0)) /\
++}          (result_0  => (max_3 = max_1)) /\
++}          (~result_0 => (max_3 = max_2));
++} modifies max;
if (x <= y) then
   ++} requires true;
   ++} ensures max_1 = y_0;
   ++} modifies max;
   max := y;
else
   ++} requires true;
   ++} ensures max_2 = x_0;
   ++} modifies max;
   max := x;
```

From this the VC generation phase generates the following LP script:

```
declare operators
   result_0 : -> Bool
   x_0, y_0, max_0, max_1, max_2, max_3 : -> Integer
..

assert
   result_0 = (x_0 <= y_0);
   max_1 = y_0;
   max_2 = x_0;
   result_0    => (max_3 = max_1);
   ~(result_0) => (max_3 = max_2);
..
```

Gory details

In the previous sections we have described the design decisions that were made and provided an overview of the algorithms used in the implementation. Here we give specific details on the actual size of the program that we have written and the time taken to write it.

The changes to the compiler took nearly 30 days to complete spread over a period of four months. Some of this time was inevitably spent without making much forward progress: the version of the Aldor compiler that we were working with contained 140000 lines of C and, although the source was well structured, we often hit dead-ends. Furthermore, we had to balance the effort required to write and debug functions to achieve our aims against

spending time to investigate whether any of the existing functions in the compiler could be tailored to our needs.

The changes that were made to the compiler are summarised below. They correspond to less than 500 lines of additional source code, including comments, blank lines and a few debugging statements! Note that an AQ file is the object language of the VC generator, a LISP representation of the annotated parse tree:

abnorm	Ensures annotations are stored correctly in the parse tree
absyn	Convert the parse tree into AQ format
axlobs	Minor additions to support generation of AQ files
emit	Minor additions to support generation of AQ files
genfoam	Skip annotations when generating intermediate FOAM code
scan	Lexical scanner extended to recognise `++}` tokens
scobind	Scoping: minor additions to support annotations
stab	Symbol table: minor additions to support annotations
syme	Symbol meanings: additions to support annotations
ti	Type inference: skip over annotations
token	Internal token to represent `++}`

The verification condition generator took six months to develop and occupies about 8500 lines of Aldor. Roughly 3600 lines are blank or nearly blank, 1600 lines are comments and 3300 lines are program statements. The source is divided into a number of different modules and the proportion of code associated with the different tasks is:

20.8%	Conversion of programs to and from AQ/LISP format
19.2%	Parsing and representation of Larch/Aldor specifications
13.8%	Annotation phase (adding suffices to identifiers)
11.1%	Low-level functions for manipulating identifiers
7.5%	Parsing of LISP expressions from text strings
5.0%	Verification condition generation
4.7%	Generation of LP scripts from VC information
4.5%	Representation and storage of identifiers
13.4%	Other functions, representation of logical expressions *etc*

5.3.4 Lessons learned

As with any programming project, one does not often make continuous program from a design through to the final implementation without taking a few wrong turnings. For example, one may decide to experiment with a new technique or idea only to decide later that it was unsuitable. In this section we summarise some of the points which may be of interest.

- Access to the source of the Aldor compiler was essential for the successful development of the prototype VC generator. Without it we would have had to write our own parser and type inferrer, which would have been infeasible. However, given more time and a faster development machine that we began with, we believe that integrating the VC generator with the rest of the compiler would provide significant benefits. We would not have to convert from the external representation of parsed Larch/Aldor programs into the internal representation of the VC generator, something that occupies nearly 30% of the source of our implementation. We would be in a much better position to perform type-inference and type-checking of Larch/Aldor annotations and one could even consider translating some annotations into runtime checks where it was feasible.

- Representing Larch/Aldor programs using a data-structure which closely matches the lexical structure of the source is not ideal. Although it allowed us to write elegant and readable code for the VC generator using recursion, the presence of unconditional branches within programs creates too many problems. With hindsight we believe that programs can be represented better using flow-graphs similar to those adopted by Vandevoorde [82]. However, our aim was to construct a prototype VC generator and we believe that our choice of data structure and analysis method were the correct ones for the task. The readability of our code, and the close coupling between the internal representation and the original Larch/Aldor program, were important during the early stages of development.

- We were surprised to discover that the most time consuming and intricate part of the whole project was associated with the annotation phase of the VC generator. This was the part of the program which adds annotations to statements that the VC generator knows about such as assignments, and which ensures that each identifier in the Larch/Aldor program is given a unique symbol each time it is mutated at runtime.

This phase takes up about 30% of the source code. If one adopts an approach which requires the state of each program variable to be given a unique logical symbol then great care needs to be taken during its design and implementation.

- Our prototype VC generator conjoins the information associated with if-then-else statements rather than splitting the context into two. Keeping a single context that records information about all possible paths significantly reduces the amount of output that the VC generator produces. It also avoids issues of how to represent and update multiple contexts and how to present them to the user. However, if the user attempts to discharge VCs that were generated after an if-then-else statement then they may have to deal with case-splits. At least this is an area in which theorem provers are good at and can provide appropriate tools for dealing with them. Although we preferred the approach of keeping different execution paths separate we found that the execution time of the VC generator and the amount of output that it produced were too large.

- Ironically we often found that our lightweight VC generator was turning into a "full" VC generator. Our design philosophy was that statements that had been annotated by the user would not be analysed any further. However, since the annotation phase was able to generate annotations for several Aldor statements (based on their semantics), there was a temptation to generate annotations for all statements. The issues of how to merge the two types of annotations might be interesting to investigate further.

- Finally, the task of converting the VCs from their internal representation into the object language of LP, the Larch Prover, was straightforward. Information obtained from the post-conditions of annotations are translated into declarations of new logical symbols and assertions that define their value. The VCs are obtained from the pre-conditions of annotations and are translated into a command instructing LP to begin a proof attempt. To assist the user, comments are used to associate assertions and VCs with the name of the source file and the line at which it appears. We believe that this type of conversion could be easily repeated for the object languages of different theorem provers such as HOL [34] or PVS [68].

5.3.5 Conclusions and future work

In the previous section we gave an overview of the current status of our VC generator; in this section we briefly review the benefits and drawbacks of the prototype and suggest directions for future work.

We have identified three main problems with our prototype:

1. The internal representation of Aldor programs is a parse tree rather than a control flow graph. While the former is ideal for analysis which is strictly lexical and which needs to be linked to the original source code (*i.e.* so that VCs can be associated with line numbers) it is less convenient for analysis which follows paths of execution. It also complicates analysis of programs which contain jumps or raise exceptions, the latter being a recent addition to Aldor.

2. Our prototype does not make any reference to the store model defined in Section 4.3 and therefore is unable to deal with concepts such as object dependency. Perhaps more importantly, the store model would allow us to make statements about the relationship between the store in the pre-state of a function and that in the post-state without needing explicit state subscripts on all identifiers.

3. Lack of type checking and type inference of annotations.

Apart from these problems we believe that the prototype has been successful in helping us to investigate the various design issues involved with constructing a lightweight VC generator. Furthermore many of these issues, such as dealing with branch points, are issues which are also relevant to traditional VC generators. The issues of type checking and type inference have not been considered very much in Larch literature to date and is touched on briefly in [85]. Our experiments with multiple executions paths produced a tool in which the output describes all paths simultaneously using guards on assertions about the program state. However, earlier versions which produced a stream of output for each individual execution path were very successful although the resulting files containing VCs and assertions were numerous and large even for the simplest of examples. The execution time for programs with only a few lines was also prohibitive.

At present the output of the prototype is in the object language of the Larch Prover and care has been taken to keep the output as readable as possible. The LP script is divided

into sections with comments which give the name of the source file and the line number associated with each symbol declaration, assertion and verification condition. Additional comments indicate whether the LP commands have been generated from annotations automatically provided by the analyser or whether they are from the user annotations. The program state is given in the form of LP assertions while the VCs themselves appear as LP `prove` commands. This means that the output from the VC generator can be stored and then passed as input to the Larch Prover. Often VCs can be proved automatically by normalisation while VCs resulting from if-then-else statements can be tackled using proof by cases. If enough information has been provided in the form of Larch/Aldor annotations, and assuming that the VCs are provable, then LP may be able to do so automatically.

Unfortunately the prototype has not been developed enough to investigate non-trivial programs. At least another month of development time would be required to extend its capabilities and to write specifications for basic data types such as records and lists. A significant proportion of the execution time appears to be spent reading and parsing the input file—the analysis and the generation of VCs does not require much computation. We believe that incorporating the VC generator into the Aldor compiler would improve the performance of VC generation to the extend that it could be comparable in runtime to the optimisation phase of the compiler.

Our experience with this prototype has lead us to consider various options for future work. Of particular interest is the possibility of including the VC generator into the Aldor compiler so that it can benefit from the type checking and type inference algorithms and can have the potential to use its control-flow graphs. In this way Larch/Aldor annotations would become an integral part of the language in a similar way to the annotations of Eiffel. There would also be opportunities for storing the annotations in the compiled libraries. Since the Aldor compiler performs a significant amount of inlining from pre-compiled libraries there is the issue of “inlining specifications” and how this might affect their meaning (if the optimiser is correct then their meaning ought to be unchanged). Indeed the optimiser may be able to use procedure specifications to increase the performance of the resulting code as demonstrated in [82].

Other possible areas of future research include the filtering and presentation of the VCs. Users may wish to restrict the output to include only those VCs which refer to a particular symbol or a region of the program. The description of the program state which our prototype currently produces could be analysed to provide a symbolic trace of values returned

by functions or assigned to variables.

5.4 Summary

In this chapter we introduced the subject of program verification. In Section 5.2.1 we defined what we mean by partial and total correctness and verification conditions. Then in Section 5.2.2 we gave a brief overview of the traditional method of VC generation which use forwards or backwards program analysis. This was followed in Sections 5.2.3 and 5.2.5 by a description of our lightweight VC generation ideas and suggestions of how VCs might be used in practice. In Section 5.3.1 the design of a lightweight VC generator was proposed and was followed by details of our prototype VC generator for Larch/Aldor. Finally in Section 5.3.5 we reviewed the prototype and provided suggestions for future research.

As we have already mentioned in earlier chapters, many of the Larch BISLs do not have any program analysis tools associated with them. Two notable exceptions are Penelope, an interactive development and verification system for Larch/Ada [35], and LcLint [24] which performs extended static-checking of C programs. Another system which makes use of Larch BISL specifications is the Speckle compiler [82].

Our work shows how Larch BISL annotations can be used in a lightweight fashion which allow the user to obtain verification conditions without the need for a completely formal development. Although our prototype has not been developed to the stage where it could be used on large programs, we believe that future versions could be incorporated into the Aldor compiler and be at least as fast as the optimisation phase of the compiler. This is important because tools analysis tools must not be too slow otherwise they will not be used by developers.

Chapter 6

Case studies in Larch/Aldor

In this chapter we describe two case studies which centre on the use of Larch/Aldor. The first of these is based around implementations of the quicksort algorithm—starting with the background theory presented in the Larch Shared Language (LSL) an implementation is produced using the reification techniques which were described in Chapter 3. From this implementation, lightweight verification conditions are derived by hand using the same techniques that are utilised by the prototype VC generator described in Section 5.3.2. The second case study examines a trimmed-down version of an Aldor library function to scan "numbers" from text strings. We show that one of the verifications cannot be discharged because it represents a bug in the implementation which had not previously been detected.

Unfortunately the prototype verification condition generator that we described in Section 5.3.2 has not been developed to the stage where it can analyse specific functions. At the moment it works at the top-level of a program and does not descend into function bodies. Furthermore we have not written interface specifications for operators for important types such as arrays and lists so any verification conditions the prototype could generate would be somewhat limited.

6.1 Quicksort

We begin this case study by defining LSL traits to provide the necessary background theory of sorting arbitrary containers. From this we derive a Larch/Aldor implementation of the

quicksort algorithm for lists using reification (*c.f.* Chapter 3) and examine the verification conditions which can be derived from them using the lightweight verification techniques of Chapter 5. Note that we are not interested here in proving correctness of our Aldor implementation of the quicksort algorithm—instead we wish to discover out how useful verifications conditions generated from Larch/Aldor specifications can be for detecting mistakes in Aldor programs.

6.1.1 Background theory

We start by defining an LSL trait for the theory of sorting generic containers. The container must be able to maintain its contents in a particular order which means that sets and bags are not suitable but lists and arrays are. Due to restrictions of the size of the page we have split the trait into three parts—the top-level part is:

```
Sorting(E, C): trait
   includes
      SortingOps(E, C),
      StrictPartialOrder(≺, E)
   asserts
      ∀ c,c1,c2:C, e,e1,e2:E
         (c1 = genSort(c2)) == (ordered(c1) ∧ permuted(c1,c2));

         permuted(empty, c) == (c = empty);
         permuted(append(e, c1), c2) ==
            ((len(append(e, c1)) = len(c2))
            ∧ permuted(c1,remove(e,c2)));

         ordered(empty);
         ordered(append(e, empty));
         ordered(c) == ¬(head(tail(c)) ≺ head(c)) ∧ ordered(tail(c));

         remove(e, empty) == empty;
         remove(e1, append(e2,c)) ==
            if (e1 = e2) then c else append(e2,remove(e1, c));
   implies
      SortingImplications(E, C)
```

The operations which are used by this theory are:

```
SortingOps(E, C): trait
   includes
      Queue(E, C)
   introduces
      genSort:  C    → C     % General sorting operation
      permuted: C, C → Bool  % Container is a permutation of another?
      ordered:  C    → Bool  % Container is ordered (sorted)
      remove:   E, C → C     % Remove specified element if present
      __ ≺ __: E, E → Bool  % Ordering relation
```

Some implications of this theory are:

```
SortingImplications(E, C): trait
   includes
      SortingOps(E, C)
   implies
      Idempotent(genSort, C)
      ∀ c,c1,c2:C, e:E
          permuted(c,c);
          permuted(c1, c2) == permuted(c2, c1);

          (e ∈ c) == (len(c) = succ(len(remove(e,c))));
          ¬(e ∈ c) == (len(c) = len(remove(e,c)));

          genSort(empty) == empty;
          genSort(append(e,empty)) == append(e, empty);
      converts
         genSort, ordered, permuted, remove
```

Using this we define a theory for the quicksort algorithm where the quicksort algorithm works as follows: to sort a non-empty container C a single element, *pivot* is removed from the container. The resulting container (which may contain other elements equal to *pivot*) is partitioned into containers *lo* and *hi*, such that all the elements, $(e \in lo)$ satisfy the ordering $(e \prec pivot)$ and all the remaining elements $(e \in hi)$ satisfy $\neg(e \prec pivot)$. The quicksort algorithm is applied to *lo* and *hi* independently and the resulting sorted containers are combined with *pivot* to produce the sorted result.

A correctness proof of this algorithm would need to show that the result of a quicksort operation is an ordered permutation of the original container and that the algorithm terminates. Informally we note that the partitioning operation does not remove any of the elements

(except the single pivot) nor does it modify them; likewise the combination of the sorted containers does not insert additional elements or mutate them. This ensures that the sorted container is a permutation of the original one. Provided that the combination of the sorted containers preserves the ordering, the result will be sorted. An induction proof will show that the quicksort algorithm is correct—the termination condition requires the size of the partitioned containers to be smaller than the original which is ensured by the removal of the pivot element before partitioning.

```
Quicksort(E, C): trait
   includes StrictTotalOrder(≺, E), Sorting(E, C, qSort for genSort)
   introduces
      pivot:              C    → E
      filter:             E, C → C
      loPart:             C    → C
      hiPart:             C    → C
      qSort:              C    → C
      concat:             C, C → C
      { __ }:             E    → C
      __ \ __ :  C, C → C
   asserts
      ∀ c,c1,c2:C, e,e1,e2:E
         ¬(c = empty) ⇒ (pivot(c) ∈ c);

         filter(e, empty) == empty;
         filter(e1, append(e2,c)) == if (e1 ≺ e2)
               then append(e2, filter(e1,c)) else filter(e1,c);

         { e } == append(e, empty);

         concat(c, empty) == c;
         concat(c, append(e,c2)) == append(e,concat(c,c2));

         qSort(empty) == empty;
         qSort(append(e,empty)) == append(e, empty);
         qSort(c) == concat(qSort(loPart(c)),
                            concat({pivot(c)}, qSort(hiPart(c))));

         loPart(c) == filter(pivot(c),c);
         hiPart(c) == remove(pivot(c),c \ loPart(c));
   implies ∀ c,c1,c2:C, e,e1,e2:E
         ¬(pivot(c) ∈ loPart(c));
         concat(empty, c) == c;
      converts concat, \, filter, qSort, loPart, hiPart
```

Note that we have strengthened the constraint on the ordering operator $\prec$ so that provides a strict total order. This allows the results results of two different sorting operations to be compared even if these operations are both unstable. Unstable sorting algorithms are able to correctly sort a container and but may produce a different ordering from another equally valid implementation. Indeed the same unstable algorithm may produce a different ordering depending on the input data used.

```
++} uses      Quicksort(S, List(S), orderOp for ≺);
++} requires  StrictTotalOrder(orderOp, S);
++} modifies  nothing;
++} ensures   result = qSort(L);
++} where     orderOp(a : S, b : S) : Boolean == {
++}               modifies nothing;
++}           }
quickSort(S:BasicType, orderOp:(S,S)->Boolean, L:List S):List S ==
{
   -- Empty and singleton lists are already sorted.
   (#L < 2) => L; -- First exit point.

   -- Select a pivot - the first element will suffice.
   pivot := L.1;

   -- Split into three parts
   ++} ensures lo = loPart(L);
   lo := [e for e in rest(L) | orderOp(e, pivot)];

   ++} ensures mid ={pivot(L)}
   mid := [pivot];

   ++} ensures hi = hiPart(L);
   hi := [e for e in rest(L) | ~orderOp(e, pivot)];

   -- Sort the upper and lower partitions
   upper := quickSort(S, orderOp, hi);
   lower := quickSort(S, orderOp, lo);

   -- Combine the pivot and upper partition
   tmp := concat(mid, upper);

   -- ...  then combine the lower parition and return
   concat(lower, tmp);
}
```

Figure 6.1: Quicksort Implementation For Lists

6.1.2 Quicksort for lists

In this section we examine a Larch/Aldor implementation of a quicksort algorithm for sorting lists (Figure 6.1) with the aim of extracting verification conditions from it using the lightweight techniques described in Chapter 5. The program was obtained from the `Quicksort` LSL trait by using reification techniques similar to those described in Chapter 3 and so we expect that any mistakes in the algorithm would stem from mistakes in the LSL specifications used as the starting point. The reification process concentrated on the three definitions of `qSort` which could be translated almost directly into Aldor.

In the sections which follow we examine the verification conditions that have been generated. We find that in this case study the annotations and the background LSL theory provide enough information to allow the verification conditions to be discharged with relative ease. This means that a user would not be inundated with verification conditions that they might not wish to investigate further.

The `modifies` clause

Before examining the body of the function we consider first the satisfaction of the `modifies` clause—in general this may be very difficult or even impossible but luckily for this implementation it is relatively easy. The program shown below uses a number of different operators, most of which are known to be purely functional (they modify nothing that is visible to their clients). The only operations which we need to focus on are `orderOp` and of course, `quickSort` itself. Even these can be shown to modify nothing visible to the client simply by appealing to the pre-condition and `modifies` clauses of `quickSort`. The remaining statements in the function can not modify the list `L` nor any of the other parameters and the function does not refer to any free variables which would be visible to the caller. Hence it does not modify anything that is visible to the caller.

6.1.3 Verification conditions

We proceed to examine line-by-line the implementation shown in Section 6.1.2 in just the same way as the lightweight verification condition generator of Section 5.3.2 would do. At each stage of the analysis the context which describes the current state of the program may

be extended and this is highlighted in the sections below by showing the formulæ which are added. During the verification condition generation, Aldor identifiers are translated into subscripted LSL symbols—this enables us to reason about the program in a time independent way. It also allows us to refer to the (symbolic) value that an identifier has before and after each program statement has been executed. If an identifier changes value then the subscript is incremented otherwise it is left unchanged.

The semantics of Larch/Aldor are such that it only makes sense to analyse functions under the assumption that their pre-conditions are satisfied otherwise the behaviour of the function is unconstrained and any information derived from it may be useless. This means that the context is always initialised with the pre-condition of the block under analysis which in this case is the body of the `quickSort` function:

$$\mathrm{StrictTotalOrder}(orderOp_0, S_0)$$

In practice the reference `StrictTotalOrder` would be replaced by the contents of the trait definition with appropriate renaming. However, there is no need to do so here since it would be unnecessarily verbose and would provide little enlightenment.

The first exit point

The first exit point of the `quickSort` function is the statement "`(#L < 2) => L`". From this and the post-condition of `quickSort` we obtain the verification condition

$$(length(L_0) < 2) \Rightarrow (L_0 = qSort(L_0))$$

which can easily be discharged using the axioms of the LSL `Queue` trait. After processing this statement the context is extended with the assertion

$$\neg(len(L_0) < 2)$$

Extracting the pivot element

The pivot is extracted using the statement "`pivot := L.1`" which is syntactic sugar for "`pivot := apply(L, 1)`". The specification of the array operation `apply` (not

shown) requires that the index 1 lies within the bounds of the array and hence we obtain the VC:

$$(1 \geq 1) \wedge (1 \leq length(L))$$

This can be easily discharged using knowledge that $\neg(len(L_0) < 2)$ which followed from the previous verification condition and a little knowledge (*i.e.* background theory) about orderings over the integers.

Partitioning the list

The assignments to `lo`, `mid` and `hi` are annotated with post-conditions and so we do not examine them any further (the lightweight approach). Their pre-conditions are trivial (simply *true*) and the context is extended with the following assertions:

$$\begin{aligned} lo_0 &= loPart(L_0) \\ mid_0 &= \{pivot(L_0)\} \\ hi_0 &= hiPart(L_0) \end{aligned}$$

Sorting the partitions

After partitioning there are two applications of `quickSort` which assign results to the variables `upper` and `lower`. The assignment statements do not require any verification conditions to be generated but the function applications do:

$$\mathrm{StrictTotalOrder}(orderOp_0, S_0)$$

This can be trivially discharged for both applications since this assertion was the first to be placed in the context. Now using the post-conditions of `quickSort` the context can be extended with two more assertions:

$$\begin{aligned} result_0 = qSort(hi_0) \quad &\wedge \quad upper_0 = result_0 \\ result_1 = qSort(lo_0) \quad &\wedge \quad lower_0 = result_1 \end{aligned}$$

The LSL $result_i$ symbols are specification variables introduced to simplify the generation of verification conditions. Here they are represent the right-hand side of the assignment statements.

Combining the pivot and upper partition

The first recombination is "`tmp := concat(mid, upper)`" and since the specification of `concat` (not shown) has no pre-condition there are no verification conditions to be generated. The post-condition of `concat` allows the context to be extended with:

$$(result_2 = concat(mid_0, upper_0)) \wedge (tmp_0 = result_2)$$

Final combination and exit

Finally we reach the last statement of the function. The application of `concat` does not generate any verification conditions itself but the context is extended with

$$result_3 = concat(lower_0, tmp_0)$$

However, since this is an exit point of the function we generate the verification condition that the post-condition of `quickSort` is satisfied, namely that:

$$result_3 = qSort(L_0)$$

This can be discharged relatively easily since our implementation closely follows the definition of *qSort* from our LSL theory. After a little rewriting the verification becomes

$$concat(qSort(loPart(L_0)), concat(\{pivot(L_0)\}, qSort(hiPart(L_0))))$$

which is one of the axioms of the `QuickSorting` trait.

6.1.4 Summary

Even from this relatively simple program it has been possible to generate quite a few verification conditions. Of these almost all were very easy to discharge by hand but a few would

require a little effort with a proof assistant such as the Larch Prover (LP). For example, the verification condition generated from the first exit point of the program was

$$(length(L_0) < 2) \Rightarrow (L_0 = qSort(L_0))$$

Attempting to prove that this holds using LP is complicated by the fact that the obvious LSL formula which could be used to discharge it is expressed in terms of list constructors rather than in terms of the length of the list:

$$qSort(append(e, empty)) \equiv append(e, empty))$$

However, this verification condition could be easily "discharged" by inspection if we recognise that $(length(append(e, empty)) = 1) \wedge (1 < 2)$.

If this case study was extended to investigate a quicksort algorithm for sorting arrays *in situ* then many more verification conditions might be generated since the implementation is more complicated. The number of these which can be automatically discharged by a proof assistant such as the Larch Prover will depend on how powerful the background LSL theories are and how detailed the Larch/Aldor interface specifications are.

One might begin by applying a lightweight verification condition generator to an unadorned program and then examining the output. Any verification conditions which are difficult to discharge may either highlight bugs in the implementation or deficiencies in the annotations of any functions which were applied. Various points in the program could then be annotated with Larch/Aldor and a new set of verification conditions generated. The process could be repeated until the program has been specified to the desired level and in the process any bugs may be identified and eliminated.

6.2 Number scanning

The second case study is concerned with the task of converting a textual representation of a "number" into a value belonging to a particular type. At its simplest this can be regarded as the conversion of text into a value of type `Float` for example. However, the Aldor `NumberScanPackage` in the AxlLib library allows values of any type `R` to be retrieved from a text string where `R` satisfies the category `Ring`. The textual representation may contain both integer and fractional parts and may have an exponent part; it may also be written using any base or radix. Several examples are given below:

Text	Floating point value
“42”	42
“3.14159”	3.14159
“4.5e-2”	0.045
“8r15”	13 (Octal 15)
“2r111.101”	7.625 (Binary 111.101)

6.2.1 Introduction

The only export of `NumberScanPackage` is `scanNumber` which performs the extraction of a “number” from a text string. In Figure 6.2 we show a slightly simplified version of the Aldor library `scanNumber` function which scans numbers in base 10 without exponents. Extending the function to deal with these features is relatively easy but would not add much to this exposition. We have annotated interesting points in the code but have not shown functions such as `scPeekChar` (we are not concerned with their implementation here). In fact these functions are defined as macros in the Aldor library but their behaviour is just the same when treated as functions. As before with the quicksort example we have presented the function as-is rather than as part of a domain to simplify the presentation.

The number-scanning algorithm is simple—a variable called `bufpos` is used to keep track of the current character within the buffer being scanned. Initially it is placed at the first character and then moved forward passed any leading whitespace characters. If there appears to be a sign symbol ($+$ or $-$) this is noted and `bufpos` is advanced. Next the integer part of the number is scanned which must always exist even if the number is a floating point value in the range $[-1, 1]$. A check is made to see if there is a decimal point and then any fractional part is scanned. Finally the function combines the integer and fractional parts into a numerical value which is returned to the caller.

6.2.2 Verification conditions

Although our annotations of `scanNumber` and its support functions are fairly detailed they do not completely capture the behaviour of the algorithm. In particular we have not attempted to describe how the buffer is scanned except to state in various places that the

```
scanNumber(R:Ring, buf:String):R ==
{
   free bufpos:Integer := 1;
   (fraction, fracExpr) := (0, 0);

   ++} requires   true
   ++} ensures    (buf[bufpos'] ≠ SPACE) ∧ (bufpos' ≥ bufpos^)
   ++} modifies   bufpos
   while (isSpace(scPeekChar()) repeat scAdvance();

   ++} requires   (buf[bufpos] ≠ SPACE) ∧ (bufpos < length(buf))
   ++} ensures    (sign' = signOf(result) ∧ (bufpos' ≥ bufpos^)
   ++} modifies   bufpos, sign
   sign := scanSign(buf);

   ++} requires   isDigit(buf[bufpos])
   ++} ensures    (mantissa = mantissaOf(result))
   ++}            ∧(bufpos' > bufpos^)
   ++} modifies   bufpos, sign
   mantissa := scanInteger(buf);

   ++} requires   (buf[bufpos] = DOT) ∨ (bufpos = length(buf))
   ++} ensures    (fraction = fractionOf(result))
   ++}            ∧(fracExp = exponentOf(result))
   ++}            ∧(bufpos' = length(buf))
   ++} modifies   bufpos, sign
   if (isDecimalPoint(scPeekChar())) then
   {
       scAdvance();
       oldpos := bufpos;
       fraction := scanInteger(buf);
       fracExpr := bufpos - oldpos;
   }

   return mantissa + (fraction*power(10, -fracExp));
}
```

Figure 6.2: Implementation of `scanNumber`

algorithm never moves backwards through the buffer. We do not regard under-specification as a problem and one may wish to treat it as part of the development process: generate verification conditions, check them and perhaps strengthen or modify the annotations. This process can be repeated as many times as it is necessary. We have decided not to define an LSL theory which would provide the semantics of predicates which appear in the

Larch/Aldor annotations. Such a theory is not essential for this case study although the types of the predicates would be required if the Larch Prover was used for proof attempts on the verification conditions.

In the beginning...

We begin with the first line of `scanNumber` and proceed in exactly the same way as before. Initially the context is empty because the `scanNumber` function has a trivial pre-condition but after the execution of the first two statements the context is

$$\begin{aligned} \texttt{bufpos}_0 &= 1 \\ \texttt{fraction}_0 &= 0 \\ \texttt{fracExpr}_0 &= 0 \end{aligned}$$

Skipping leading whitespace

We now consider the while loop which is designed to skip past any leading whitespace characters. The pre-condition generates the trivial verification condition *true* which can be ignored but the `modifies` clause indicates that `bufpos` *might* be changed. This means that we need to introduce a new LSL symbol for it, $\texttt{bufpos}_1$ which corresponds to the post-state of `bufpos`. The post-condition allows us to extend the context with

$$\begin{aligned} \texttt{buf}_0[\texttt{bufpos}_1] &\neq \texttt{SPACE} \\ \texttt{bufpos}_1 &\geq \texttt{bufpos}_0 \end{aligned}$$

Note that we do not know what the concrete value of $bufpos_1$ is but we do have a relationship between it and its original value $bufpos_0$. Also note that we refer to the parameter `buf` as $\texttt{buf}_0$: since it is never modified it will always be referred to as this.

Reading the sign, mantissa and fraction

Moving past the while loop we reach the statement which scans the sign of the number if it is present. This has a more complicated pre-condition which needs a little tidying up before it can be renamed into an LSL symbol. As described in Section 5.3.3, unadorned

occurrences of an identifier such as `bufpos` which appear the pre-condition are a shorthand for the identifier in its pre-state. Since the pre-state for this identifier is $\texttt{bufpos}_1$ the VC (in the context of the preceding sections), becomes:

$$(\texttt{buf}_0[\texttt{bufpos}_1] \neq \texttt{SPACE}) \wedge (\texttt{bufpos}_1 < \mathit{length}(\texttt{buf}_0))$$

which simplifies to $\texttt{bufpos}_1 < \mathit{length}(\texttt{buf}_0)$. This verification condition highlights a problem with the implementation which assumes that `buf` is not the empty string and does not contain just whitespace. Using this information we may wish to add an appropriate test to ensure that these two conditions are dealt with correctly.

The post-condition extends the context with

$$\begin{aligned} \texttt{sign}_1 &= \mathit{signOf}(\mathit{result}) \\ \texttt{bufpos}_2 &\geq \texttt{bufpos}_1 \end{aligned}$$

where result is the LSL symbol representing the result of the function that we are analysing: `numberScan`. The signOf predicate would be defined in an LSL theory just as was done in the quicksort case study.

The analysis of the statements which read the mantissa and the fractional part of the number is similar to that for reading the sign. We generate the verification conditions

$$\begin{aligned} &\mathit{isDigit}(\texttt{buf}_0[\texttt{bufpos}_2]) \\ &(\texttt{buf}_0[\texttt{bufpos}_3] = \texttt{DOT}) \quad \vee \quad (\texttt{bufpos}_3 = \mathit{length}(\texttt{buf}_0)) \end{aligned}$$

and extend the context with

$$\begin{aligned} \texttt{mantissa}_1 &= \mathit{mantissaOf}(\mathit{result}) \\ \texttt{bufpos}_3 &\geq \texttt{bufpos}_2 \\ \texttt{fraction}_1 &= \mathit{fractionOf}(\mathit{result}) \\ \texttt{fracExp}_1 &= \mathit{exponentOf}(\mathit{result}) \\ \texttt{bufpos}_4 &= \mathit{length}(\texttt{buf}_0)) \end{aligned}$$

The final statement

The last line of the program is what the algorithm has been building up to—this is the point where the partial results are combined to produce a value in the domain R. However, there

is a problem: although the `power` export from the category `Ring` does not have a pre-condition, implementations of it in specific domains might do. For example, domains such as `Integer` require the exponent to be positive. Hence the VC when $(\texttt{R} = \texttt{Integer})$ is:

$$-\texttt{fracExp}_1 \geq 0$$

If we analyse the function without the annotation on the `if` statement which scans the fractional part of the number then we find that if there is no fractional part then

$$\texttt{fracExp}_1 = \texttt{fracExp}_0$$

otherwise if there is a fractional part then

$$\texttt{fracExp}_1 \geq 0$$

This means that if the number being scanned has a fractional part, and $(\texttt{R} = \texttt{Integer})$, then the verification condition cannot be satisfied.

Clearly this is a bug in the implementation and indeed a small test program demonstrates it: when $(\texttt{R} = \texttt{Integer})$ the program stops with a runtime error because `power` from `Integer` detects the negative exponent. However, when $(\texttt{R} = \texttt{SingleInteger})$ (machine-precision integers) `power` quietly returns the wrong value which causes `scanNumber` to return the wrong value.

6.2.3 Summary

In this case study we found a few simple verification conditions, none of which could be discharged automatically. The annotations could be strengthened to describe the format that the text string must take and how the algorithm would scan it which may help. However, we discovered from the first verification condition that the function was not checking to see if it had reached the end of the buffer. We also discovered a more serious bug in the final statement of the program which prevents integers from being scanned correctly when the text string represents a number with a fractional part after the decimal point.

This latter discovery highlights a problem with the generation of verification conditions from polymorphic functions. The `scanNumber` function is parameterised by the domain

`R` where `R` satisfies the category `Ring` and we have found that different verification conditions can be generated depending on the concrete value of `R`. This is a problem with the implementation which is relying on properties of `R` which are not described by its category.

From the lightweight verification condition generation stand-point it means that some verification conditions can only be generated when the function has been specialised with a particular `R`. This is an issue which can not be easily avoided unless the body of each polymorphic function is analysed every time it is applied. This goes against the basic principles of lightweight verification condition generation and eliminates all the benefits of modularity that we rely on for creating easily comprehensible output. We return to this point in Section 6.3.

6.3 Conclusions

In this chapter we looked at two different programs. The first was a function which sorted lists of values using a user-supplied ordering and the quicksort algorithm while the second was a function for scanning numbers from text strings taking into account a possible fractional part.

For the quicksort we began by defining an LSL theory of sorting contains which can retain their elements in a specified order. This was extended to describe the quicksort algorithm and then, using the reification techniques of Chapter 3, a simple implementation was produced and annotated using Larch/Aldor. The implementation was analysed from top to bottom to generate verification conditions using the lightweight techniques described in Chapter 5. Most of these verification conditions were easy to discharge automatically although one or two would require a little effort using the Larch Prover, LP.

The number scanning example was presented *as is* with no background theories described in LSL or otherwise. The implementation was a cut-down version of a function which is part of the Aldor AxlLib library. To simplify the presentation support for arbitrary bases and exponents were removed and the behaviour of the the main components were annotated with Larch/Aldor specifications. As with the quicksort case study, the function was analysed from the start to the end. This time fewer verification conditions were generated and none could be automatically discharged. This was mainly due to the complexity of

describing the format that the text string being scanned ought to have and the behaviour of the character-by-character scanning. However, we discovered two mistakes in the implementation: the first was that there were no safety checks to ensure that the scan did not reach the end of the text string prematurely. The second was more serious and depended on the specification of the `power` export—this export is defined in the domain `R` which was passed as an argument to the scanner.

This raised an interesting question about how polymorphic functions such as `scanNumber` ought to be analysed. Using the lightweight techniques that we described in Chapter 5 we would be unable to generate a verification condition which would highlight the bug. It was only after considering whether the verification conditions for `scanNumber` specialised to a particular domain such as `R = Integer` that we realised there might be a problem. As it stands, the implementation is incorrect since it depends on a "specification" of `power` which is not the same for all domains belonging to the category `Ring`.

It is this kind of subtle mistake that a verification condition generator ought to be able to detect. Unfortunately it is not clear to the author what the best way to achieve this is. One method would be to analyse the body of a polymorphic function each time it is applied in a context where the values of the abstract type parameters are known. This option is not very good since it destroys the modularity that we have been utilising to achieve the "lightweight" technique. An alternative might be to store "pending" verification conditions along with the compiled version of a function. For the number scanning example the pending verification conditions would include the fact that the pre-condition of `power` exports must be satisfied under the context which was constructed during the original analysis. Now whenever the analyser encounters the application of a library function it attempts to discharge the pending verification conditions.

During both case studies we also discovered the potential for incremental development of interface specifications. Initially the unadorned version of the program could be analysed and important verification conditions examined. Then parts of the program could be annotated, perhaps directly as a result of looking at the verification conditions generated previously. This process could be repeated for as long as necessary perhaps producing a completely annotated program. The disadvantage of a program in which every statement is annotated is that changes to the implementation will not be detected by the lightweight analysis which will only look at the annotations.

We believe that there is a case for allowing the lightweight analysis to descend a little deeper than it does at present. For example in `scanNumber` it might be useful to check that the while loop really does ensure that the current character in the scan is not a space. Quite how to achieve this without losing the benefits of our lightweight analysis is not clear. One possibility is to consider the verification conditions as labeling a tree structure. At the root or the first level of the tree would be all the verification conditions associated with the normal lightweight analysis of the program. Below this are the results of applying to lightweight technique recursively to each statement with the intention of showing that they satisfy their interface specification.

The analysis would still not descend through function applications and would just apply to the body of a specific function. The depth of the tree and the number of nodes which are computed ought to be performed lazily especially since the user may only be interested in the very top level. This would require a good user-interface to make it possible, perhaps a graphical user-interface may be essential. One could envisage the user's program represented where the bodies of annotated statements can be folded away and hidden from view. The user would be able to unfold specific statements as far as as they wish and the analyser would generate verification conditions for all unfolded nodes.

Chapter 7

Conclusions

The motivation for our work originated from the desire to provide support for the construction of reliable computer algebra programs and to allow existing implementations to be checked to ensure that their component functions are only applied in a correct manner. To this end we have examined the use of VDM reification techniques [50] to provide a safe translation from specifications to implementations, we have designed a Larch annotation language for Aldor, the compiled extension language of the **axiom** computer algebra system (Chapter 4) and we have developed a methodology of lightweight program verification (Chapter 5). We have also constructed a prototype tool for the automatic generation of verification conditions from Larch/Aldor programs and demonstrated how such verification conditions might be utilised (Chapter 6).

7.1 Software development for CAS

We believe that the components of modern computer algebra systems such as Maple [13], **axiom** [48] and Mathematica [86] generally perform their tasks correctly. The mathematics upon which they are based may have been studied for many years, sometimes even before computers and CAS were invented. These components may be exposed to the user/developer as library functions which can be used to extend the original system. However, although the components themselves may be correct, programs constructed from them may not be. For example, assumptions about the behaviour of the components might

be invalid or side-conditions might not be satisfied. In Section 6.2.2 we highlight a bug which appears to arise from an invalid assumption about the behaviour of the exponentiation operator—the assumption is not valid in every domain that the implementation claims to accept.

7.1.1 Reification

We encourage the use of reification (see Chapter 3) to assist with the construction of new implementations and modification of existing ones. This technique allows specifications to be transformed into other specifications and, eventually, implementations. Each transformation need to be justified, ideally by providing a retrieval function or relation which allows the original specification to be obtained from the new one. The transformations are generally performed with the aim of obtaining a specification that is close to the chosen implementation language. Thus one may begin with a very abstract specification written in terms of generic contains and end up with a specification in terms of linked lists. The operations defined by the specification may also be transformed, again with the intention of obtaining something that can easily be implemented. In Chapter 3 we also show how this technique can be used to produce successively more efficient (faster) implementations.

However, without effective tool support this technique may not scale very well; even with support (*e.g.* the RAISE system [16, pages 101–102]) the burden of justifying each transformation step may be too great for large programs [16, page 102]. Despite the scalability problems we still believe that it has a place in software development—even if an entire system is not developed in this way, individual components could benefit considerably. In fact each transformation step need not be justified with complete rigour—an informal argument may help to identify potential problems or issues with less effort. A successful system based on similar ideas has been developed by the High Assurance Team at ICL [53]. Their ProofPower tool allows functional requirements written in Z notation [72] to be checked and refined before being prototyped in Compliance Notation. The prototype implementations can be subsequently refined into Ada programs, possibly generating verification conditions along the way. Like us, they adopt a pragmatic approach and allow verification conditions to be discharged by formal or informal arguments as required.

7.1.2 Proving properties of specifications

Once specifications have been written then attempting to prove properties about them can be valuable. Not only does it provide a way to detect inconsistencies and under-specification but issues which may affect the design significantly could be identified. It has been pointed out by Brooks [8] and others that the sooner problems are identified the easier it is to deal with them—in a commercial environment this is particularly important. One may also learn more about the particular problem domain and potential optimisations may become apparent. Even the analysis of a simply binary search procedure can identify redundant cases in a naïve implementation—if the program spends a lot of time in the search procedure then such observations may be very important.

7.1.3 Annotating source code

One of the main contributions of this thesis is the design of a Larch annotation language for the Aldor computer algebra programming language and central to the Larch philosophy is the use of these annotation languages for the clear, concise and unambiguous description of the behaviour of functions and procedures. As part of our lightweight approach to formal methods and program verification we allow users to reason about certain aspects of their programs before they have been completely implemented. If a top-down development strategy is adopted then low-level procedures may be defined as stubs and annotated with their intended behaviour. By investigating verification conditions generated from such programs the developer may decide to modify the specifications of the low-level functions. This process can be iterated until a suitable implementation can be produced and the implementation itself can be checked against its specification.

These lightweight techniques can also be applied to legacy code in a similar way to that discussed by Evans in [24].

7.1.4 Verification conditions

As we point out in Section 5.2.5, it may not be feasible to prove whether a given verification condition is true or false. Thus we adopt a pragmatic approach and allow the user/developer

to decide on the best way to use them. They may decide to employ a mechanical tool such as an automated theorem prover, their specialist knowledge may be used to convince themselves of the validity or otherwise of verification conditions, or they may simply wish to trust them. Verification conditions might also be fed back into the specifications as extra constraints that must be satisfied by the implementation.

7.2 Contributions of this research

Our research provides two main contributions—the first is a new Larch annotation language for the Aldor programming language and the second is the methodology of lightweight program verification. In Chapter 4 we defined the syntax of Larch/Aldor and wrote a model of its store using the Larch algebraic specification language called LSL. Our annotation language goes a step further than many of the Larch languages by allowing arbitrary statements in the programming language to be annotated, for example allowing the specification of loop invariants. We have also considered the issue of functions-as-parameters and, in the case studies of Chapter 6, the issue of polymorphic functions.

Chapter 5 describes our technique of lightweight program verification. Verification conditions may be generated from programs written using Larch/Aldor either by hand or automatically. We have produced a prototype lightweight verification condition generator written in Aldor which has been shown to work successfully on small programs. Due to lack of time we have been unable to develop the necessary background theory to allow types more complicated than the integers to be reasoned with. It is this constraint that restricts the size of program which can be analysed rather than anything else.

In Section 5.2.3 we describe how our technique differs from “traditional” Hoare [42] style of program verification and in Section 5.2.5 we suggest various ways in which verification conditions can be used.

Particular limitations include the analysis of polymorphic functions and providing user-control over which parts of a program are examined by our tool. In general, verification conditions for polymorphic functions can only be investigated when type parameters are substituted for concrete values. We are not sure of the best solution for this problem—perhaps pending verification conditions need to be associated with functions in addition to

any annotations. The problem of the user more control over which parts of their program is analysed by the lightweight verification condition generator is also difficult. This is an HCI (human-computer interaction) problem and we envisage that a graphical user interface may play an essential part in its solution.

7.3 Future work

One of the main drawbacks of the Larch approach at the time of writing is that it is based on first-order logic. Although this is sufficient for many programs, those involving higher-order constructs and polymorphism may present problems which would be better dealt with using a higher-order logic. Furthermore the tool support for reasoning about Larch specifications is limited to a proof-assistant which does not appear to be developing any further. If we were to start afresh then a system such as PVS [68] or HOL [34] might prove to be beneficial. These systems are popular, stable and are being actively developed.

Restrictions imposed by time taken to recompile the Aldor compiler following our modifications (using the fastest machine available to us), played a significant part in the choice of architecture for the lightweight verification condition generator (see Section 5.3.2. Instead of embedding the tool inside the compiler where facilities such as type inference and library access were easily available, we were forced to make minor changes to the compiler to support an external analyser. The compiler already provided a way to translate its internal data structures into a LISP-like object and this was augmented with type and specification annotations to assist the external tool. Three years further on, machine speeds have increased considerably and it is now possible to make adjustments to the compiler interactively—compilation times can be measured in seconds and minutes instead of hours. Thus it would be feasible to incorporate the technology of verification condition into the compiler and to benefit from all the features that the compiler provides. The result would be faster analysis of program annotations and the annotations (and possibly verifications conditions as well) could be easily stored in Aldor libraries along with the implementations.

With regards to the verification condition generator itself, more work needs to be done to strengthen the link between our store model of Chapter 4.3 and its use in our implementation. This will increase the expressiveness of verification conditions which will be able to refer to arbitrary program states instead of just the current state. In addition, the

model itself could be extended to provide better support for various features of the Aldor language and the specification of basic Aldor domains needs to be undertaken. This will enable developers to make use of the large body of specifications of **axiom** domains that has been developed by Kelsey [52]. As mentioned in Section 6.3, the verification condition generator needs to be extended to deal with issues such as polymorphic functions and pending verification conditions, and to allow the user better control over which parts of their programs are analysed.

Appendix A

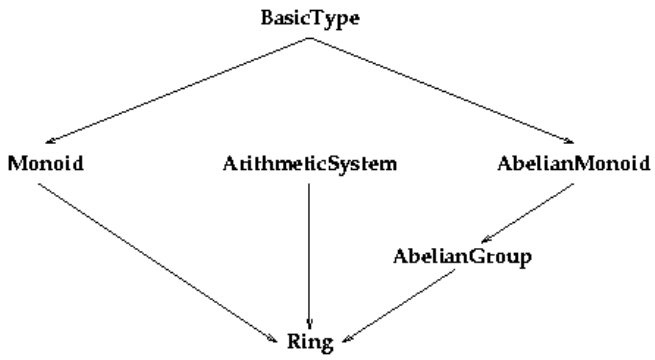
Introducing Aldor

In this section we provide a more thorough introduction to Aldor, the programming language which we are designing our Larch BISL for. We describe the language in more detail than we did in the introductory chapter but we refer readers to [83] for a more thorough description. We begin in Section A.1 by describing Aldor categories and follow this with a description of domains in Section A.2, a look at functions in Section A.3 and a brief study of some other interesting features in Section A.4.

A.1 Categories

An Aldor category is a collection of declarations parameterised by zero or more values (usually domains). They are closely related to Haskell type classes [44] and Java interfaces. Categories specify information about domains and allow functions to place restrictions on the types of domain which they can use. They usually contain function declarations which may be conditional; default values may be defined where appropriate. Although categories may be used as values they are often associated with constants whose names are intended to convey the meaning of the category. Thus a domain which claims to satisfy the `AbelianMonoid` category is expected to support additive arithmetic while a domain satisfying `Finite` is expected to only have a finite number of values. The semantics conveyed by these names cannot be inforced by a compiler and this is a major motivation for the design of Larch/Aldor and creation of associated static-checking tools.

Multiple inheritence of categories is supported and is used to derive new categories from simpler ones. It also enables a hierarchy to be constructed—for example, the `Ring` category hierarchy from one of the Aldor libraries can be represented pictorially:



Below is an example of an Aldor category. The `%` symbol represents the domain (or type) which will implement the category; the `with` clause defines the category value:

```
SetCategory(T:BasicType) : Category == with {
   -- Inherit equality tests, output operators etc.
   FiniteLinearAggregate(T); -- Inheritence

   -- Standard set operations.
   empty        : ()     -> %; -- Create an empty set
   union        : (%, %) -> %;
   intersection : (%, %) -> %;
   difference   : (%, %) -> %;

   -- Useful test operations.
   member?      : (T, %) -> Boolean;
   subset?      : (%, %) -> Boolean;
   superset?    : (%, %) -> Boolean;

   if (T has Order) then {
      smallest : % -> T;
      largest  : % -> T;
   }

   default { -- Default implementations
      subset?(s1:%, s2:%) : Boolean ==
         (intersection(s1,s2) = s2) /\ (s1 ~= s2);
      superset?(s1:%, s2:%) : Boolean ==
         (intersection(s1,s2) = s1) /\ (s1 ~= s2);
   }
}
```

Firstly note that this category is parameterised by the domain `T` which has type `BasicType`. This means that the category value can only be created if the supplied domain `T` provides all the operations defined by the category `BasicType`. This in turn means that when writing `SetCategory` we can rely on the fact that all the operations of `BasicType` are available to us (such as equality tests and a basic output operator). In the body of the category the exports of `FiniteLinearAggregate(T)` are inherited which means that we do not have to define them ourselves. Next we see four set constructors followed by three tests. If the domain `T` has an ordering defined on its elements then it ought to satisfy the category `Order`. Thus whenever `T` satisfies `Order` our category knows that the operations `smallest()` and `largest()` are meaningful and it marks their declarations as being conditional on "`T has Order`". Finally the default implementation of the `subset()` and `superset()` operators can be defined in terms of the `intersection()` and equality test operators. Any domain which satisfies this category may choose to define different implmentations for these operators.

A.2 Domains

An Aldor domain is an environment which provides a collection of exported constants and definitions for any declarations in the categories it claims to satisfy. Like categories, domains may be parameterised by arbitrary values including other domains; unlike categories only single inheritence of domains is permitted provided that the representations of the parent and child are compatible. An unusual feature of Aldor is that domains may be extended *post facto* to provide additional exports. This enables libraries to define a basic interface which can be enhanced by other libraries later on.

For convenience domains are often divided into two classes—those which define a distinguished type are referred to as *abstract datatypes* while those which do not are known as *packages*. Packages are useful for grouping together functions and other named values which may be imported into the current scope as a single unit. Those which export two or more types can be used to implement multi-sorted algebras as shown in the example below [83, page 88]. The package `NumberSorts` exports the types `Nat` and `Rat`, along with three operators to work over values of these types:

```
NumberSorts == add {
   Nat == Integer;
   Rat == Ratio(Natural);

   rat(a:Nat, b:Nat): Rat == a / b;
   num(r:Rat):Nat == numer r;
   den(r:Rat):Nat == denom r;
}
```

Abstract data types are defined in Aldor using the same syntax as packages. The only difference is that a distinguished type (represented by the symbol `%` in the domain definition) is exported along with operations to work over values of that type. Aldor provides two macros called `per` and `rep` to convert between the internal and external representation of domain values to hide the low-level type coercion involved. In the example below the `SimpleNat` abstract data type has two exports corresponding to the constant zero and the successor function. This time we have specified the category which this domain satisfies:

```
SimpleNatCat : Category == with {
   0    : %;          -- A constant representing zero
   succ : % -> %;     -- The successor operation
}

SimpleNat : SimpleNatCat == add {
   Rep == Integer;    -- Integers as the internal rep
   import from Rep;

   0 : % == per 0;    -- Type inference handles overloading
   succ(n : %) : % == per ((rep n) + 1);
}
```

The exports of a domain are *constants* whose values are defined when the domain itself is defined. Variables which are declared at the top level of a domain are local to a particular domain instance. Single inheritance between domains is achieved by writing the name of an existing domain on the left hand side of the `add` expression. The representation of the parent domain must be compatible with that of the inheriting domain—for packages this is not a problem since they have no representation while abstract data types generally use their parent as the representation. An example of category and domain inheritance can be seen together in the example below:

```
ComplicatedNat : SimpleNatCat with {
   pred : % -> %;
} == SimpleNat add {
   Rep == SimpleNat;

   pred(n : %) : % ==
      if (n = 0) then 0 else per ((rep n) - 1);
}
```

The `ComplicatedNat` domain provides all the exports of `SimpleNatCat` (without needing to provide definitions for them) as well as a function to compute the predecessor of a natural number. In practice it would be better to extend the `SimpleNat` domain rather than use inheritance (see Section A.4.3). Note that the only operations which are exported are those which satisfy the category. This mechanism provides domain inheritance with total control over the interface and allows domains to be extended or restricted as appropriate.

A.3 Functions

Functions in Aldor are essentially the same as functions and procedures in any other imperative language. However, the syntax of the Aldor type system often makes function definitions simpler and easier to understand than those in C for example. The argument list and return values of a function are declared as tuples of type `Type` and since elements of tuples can be named or assigned default values, so can function parameters. The syntax for declaring functions has already been shown in Section A.1 as part of category values and need not be shown here.

As with categories and domains, functions may be parameterised by any value including other functions and types. Function values (anonymous functions) can be created in a similar manner to category and domain values but the syntax for this is a little obscure. Currying and recursive function definitions are also possible. Functions may use dependent types and this can be used to provide support for parametric polymorphic procedures and domain manipulating functions as shown below:

```
-- Sum the elements of a list: the constant 0 and the
-- addition operator + are exported by any domain which
-- satisfies Ring. The third argument defaults to 0 if
-- it is omitted.
sumList(R:Ring, l:List(R), seed:R == 0):R ==
{
   -- The # operator obtains the length of the list 'l'.
   if (#l = 0) then
      seed;
   else  -- Recurse for brevity of example.
      first(l) + sumList(R, rest(l), seed);
}

-- A fairly convoluted function declaration!
Ladder: (D: with {f: % -> E}, E: with {g: %-> D}) -> Type;
```

In the example above, the type of the second argument of the `sumList` function clearly depends on the value of the first argument, `R`. The third argument has a default value of `0` and may be omitted when the function is applied. The declaration of the `Ladder` function uses mutually dependent types and was taken from [83, page 82].

A.4 Other features of Aldor

In this section we examine three other features of Aldor which are not often found in other imperative programming languages. These are generalised iterators which are known as generators, fluid variables and *post facto* extensions.

A.4.1 Generators (coroutines)

When programming in a language which has a wide variety of aggregate data types there is a problem over the choice of algorithm for iterator over all the elements of an object. For example, an efficient list traversal algorithm is likely to be less inefficient when applied to an array and *vice versa*. In Aldor, the solution to this problem is to use generators from which values can be obtained serially. A domain which implements an aggregate data type will export a function `generate` which returns a generator designed to efficiently iterate over elements of the aggregate. In fact since generators are values they are not restricted to being used to iterate over data strucutres. An example of a simple generator value is:

```
generate { yield 1; yield 7; yield 3; yield 8; }
```

The first time a value is requested from the generator defined above the value 1 will be returned, the next time 7, 3 and finally 8. After this point the generator is empty and no more values can be extracted from it. As one might expect, generators are often used to provide values for `for` loops—while the generator is not empty, assign its next value to the loop variable and execute the instructions in the body of the loop:

```
for i in generate(0..10) repeat
   foo(i);
```

In this example the integer segment $(0 \ldots 10)$ is converted into a generator which is used to assign values to the loop variable `i`. The loop terminates when the generator is empty. Since this use of a generator is so common Aldor allows the explicit application of the `generator()` function to be omittted. We can also iterate over multiple generators in parallel with the loop terminating as soon as one of the generators becomes empty:

```
for i in 0..10   for j in 100.. repeat
   foo(i, j);
```

Some more interesting generators can be seen in the following example:

```
-- An infinite stream of zeros
Zeros:Generator Integer == generate { repeat yield 0; }

-- An infinite stream of even numbers
Evens:Generator Integer == generate {
   for i in 0.. repeat
      if (i rem 2) = 0 then yield i;
}

-- Pre-order tree walker
PreOrder(tree:BinaryTree T):Generator T == generate {
   if (not empty? tree) then {
      yield node tree;
      for n in PreOrder(left tree) repeat yield n;
      for n in PreOrder(right tree) repeat yield n;
   }
}
```

The first generator consists of a loop which never terminates and whose body is a single yield. Thus it will always produce the value 0 however many times it is used. The second generator uses a `for` loop, which itself uses a generator over the open-ended segment of integers starting at zero. The test ensures that only even values are obtained. Finally `PreOrder` is a function which traverses a binary tree in pre-order and converts it into a generator—note the recursive definition.

From these examples it can be seen that generators are not simply lists or infinite sequences. Once a value has been returned by the generator then its execution is suspended until the next value is requested. Thus generators could conceivably be used to model concurrency at a primitive level.

A.4.2 Fluid variables

A fluid variable is a variable which has dynamic rather than lexical scope. Such variables exist throughout the lifetime of a program and may be rebound in any lexical scope. If rebinding takes place then the original value will be restored once execution leaves that lexical scope level. This facilty can be very powerful if used correctly. Unfortunately traditional local program analysis is not sufficient to determine whether or not a fluid variable has been assigned an initial value, thus use-before-definition errors can easily occur unless they are used with caution. An example of the use of fluid variables is given below based on an example in [83, page 115]:

```
fluid n:Integer := 27;

f():() == print << "n = " << n << newline;

g():() == {
   fluid n := 42; -- Re-binding.
   f();
}

f();   -- Prints "n = 27" (original value)
g();   -- Prints "n = 42" (using temporary binding)
f();   -- Prints "n = 27" (back to original binding)
```

A.4.3 *Post facto* extensions

A problem which often arises during the re-use of libraries in object-oriented systems is how to extend existing types or classes with new operations. This usually happens when a type needs to be used in a situation for which it was not designed, even though there is no reason why it ought not to be.

Consider, for example, a library which provides operations for transmitting data over a binary channel. When using a language such as C++ we will need to provide operations to convert values of each data type that we are interested in two and from a format which is suitable for transmission. These routines could be defined in a class called `Flatten` which is made available to clients of the communications library. However, if we wish to transmit values from a data type which is not supported by `Flatten` then we will need to define a new class, `ExtendedFlatten`, which provides all the operations of `Flatten` along with our extensions. At the same time another developer may define their own `ExtendedFlatten` class to support a different set of data types. Anyone who needs to use both extensions may need to define yet another class and so on.

Aldor provides an alternative in the form of *post facto* extensions. An existing domain or domain-valued function can be extended at any point to provide additional exports. In the example above, `Flatten` could be extended with operations to convert values to and from any domain we know about. However, a better solution would be to extend the domains of interest with operations such as `pack()` and `unpack()`. Now the communications library can be defined to use values of any domain which exports these operators. Note that unlike in Aldor, in a language like C++ this approach suffers from even more problems than before since new classes would need to be defined for each of the data types of interest. For example, the class `PackableInteger` could inherit from the class of integers and provide the packing and unpacking operations. As before, problems arise when a different developer creates a class such as `DifferentiableInteger` to solve a similar but unrelated problem. Combining the two might create a class with a name such as `DifferentiablePackableInteger` or `PackableDifferentiableInteger`.

Extensions are frequently used in the construction of Aldor libraries. Initially simple implementations of domains such as machine-precision integers (`SingleInteger`) are provided. These are later extended to their full functionality and can be extended further by the user as required. In the example below the library domain of text strings is extended to

provide a '+' operation as a shorthand for string concatenation.

```
extend String : with {
    + : (%, %) -> %;
} == add {
   (a:%) + (b:%):% == concat(a, b);
}
```

Note that the syntax is similar to that used to define a domain (see Section A.2. The extended domain will satisfy all the categories the extendee satisfied in addition to the categories which are specified by the extension.

Appendix B

Reification—source code

In this appendix we give the **axiom** source code each of the implementations that were described in Chapter 3. Please refer back to that section for the meaning of each function and the mathematics which lie behind it. We have chosen not to include the interface specifications with the source code.

B.1 Level 1 implementation

This **axiom** program evaluates Equation 3.14 (see Section 3.1.2) using Equation 3.10.

```
Fab: (PI, PI)           -> Expression(Integer)
Rab: (PI, NNI, PI, NNI) -> Expression(Integer)
Rnl: (PI, NNI)          -> Expression(Integer)
Nnl: (PI, NNI)          -> Expression(Integer)

Fab(na,nb) ==
   na2    := na**2
   nb2    := nb**2
   na4    := na**4
   result := 0::Expression(Integer)
   for la in 0..(na - 1) repeat
      for lb in 0..(nb - 1) repeat
         if (abs(la - lb) = 1) then
            rab    := Rab(na,la,nb,lb)
            result := result + max(la,lb) * rab**2
   ((nb2 - na2)/(3*nb2*na4)) * result

Rab(na,la,nb,lb) ==
   expr := Rnl(na,la) * r**3 * Rnl(nb,lb)
   integrate(simplify(expr), r=0..%plusInfinity)

Rnl(n,l) ==
   Nnl(n,l) * (2*r/n)**l * exp(-r/n) * laguerreL(2*l+1,n+l,2*r/n)

Nnl(n,l) ==
   numerator   := factorial(n - l - 1)
   denominator := 2*n*factorial(n + l)**3
   sqrt((2/n)**3 * (numerator/denominator))
```

B.2 Level 2 implementation

This **axiom** program evaluates Equation 3.14 (see Section 3.1.2) using Equation 3.16.

```
Fab: (PI, PI)           -> Expression(Integer)
Rab: (PI, NNI, PI, NNI) -> Expression(Integer)
Qnl: (PI, NNI)          -> Expression(Integer)

Fab(na,nb) ==
   na2    := na**2
   nb2    := nb**2
   na4    := na**4
   result := 0::Expression(Integer)
   for la in 0..(na - 1) repeat
      for lb in 0..(nb - 1) repeat
         if (abs(la - lb) = 1) then
            rab    := Rab(na,la,nb,lb)
            result := result + max(la,lb) * rab**2

   ((nb2 - na2)/(3*nb2*na4)) * result

Rab(na,la,nb,lb) ==
   alpha := (na + nb) / (na * nb)
   expr  := Qnl(na,la) * Qnl(nb,lb) * exp(-alpha*r) * r**3
   integrate(expr, r=0..%plusInfinity)

Qnl(n,l) ==
   term1 := r**l
   term2 := (2/n)**((2*l + 3)/2)
   numer := factorial(n - l - 1)
   denom := 2*n * factorial(n + l)**3
   term3 := sqrt(numer/denom)
   term1 * term2 * term3 * laguerreL(2*l+1,n+l,2*r/n)
```

B.3 Level 3 implementation

This **axiom** program evaluates Equation 3.14 (see Section 3.1.2) using Equation 3.21.

```
Fab: (PI, PI)           -> Float
Rab: (PI, NNI, PI, NNI) -> Float
Qnl: (PI, NNI)          -> UnivariatePolynomial(r,Float)

Fab(na,nb) ==
   na2    := na**2
   nb2    := nb**2
   na4    := na**4
   result := 0.0
   for la in 0..(na - 1) repeat
      for lb in 0..(nb - 1) repeat
         if (abs(la - lb) = 1) then
            rab    := Rab(na,la,nb,lb)
            result := result + max(la,lb) * rab**2
   ((nb2 - na2)/(3*nb2*na4)) * result

Rab(na,la,nb,lb) ==
   alpha  := (na + nb) / (na * nb)
   poly   := Qnl(na,la) * Qnl(nb,lb) * (r**3)::UP(r,Float)
   coeffs := vectorise(poly, degree(poly) + 1)::List(Float)
   result := 0.0
   for Ci in coeffs     for i in 0.. repeat
      result := result + Ci*factorial(i)/(alpha**(i + 1))
   result

Qnl(n,l) ==
   term1 := r**l
   term2 := (2.0/n)**((2*l + 3)/2)
   numer := factorial(n - l - 1)
   denom := 2.0*n * factorial(n + l)**3
   term3 := sqrt(numer/denom)
   term1 * term2 * term3 * laguerreL(2*l+1,n+l,2*r/n)
```

B.4 Laplace 1 implementation

This **axiom** program evaluates Equation 3.14 (see Section 3.1.2) using a Laplace transform.

```
Fab: (PI, PI)           -> Expression(Integer)
Rab: (PI, NNI, PI, NNI) -> Expression(Integer)
Tnl: (PI, NNI)          -> Expression(Integer)
Nnl: (PI, NNI)          -> Expression(Integer)

Fab(na,nb) ==
   na2    := na**2
   nb2    := nb**2
   na4    := na**4
   result := 0::Expression(Integer)
   for la in 0..(na - 1) repeat
      for lb in 0..(nb - 1) repeat
         if (abs(la - lb) = 1) then
            rab    := Rab(na,la,nb,lb)
            result := result + max(la,lb) * rab**2
   ((nb2 - na2)/(3*nb2*na4)) * result

Rab(na,la,nb,lb) ==
   expr   := Tnl(na,la) * r**3 * Tnl(nb,lb)
   result := laplace(expr,r,s)
   eval(result,s = (na + nb)/(na*nb))

Tnl(n,l) ==
   Nnl(n,l) * (2*r/n)**l * laguerreL(2*l+1,n+l,2*r/n)

Nnl(n,l) ==
   numerator   := factorial(n - l - 1)
   denominator := 2*n*factorial(n + l)**3
   sqrt((2/n)**3 * (numerator/denominator))
```

B.5 Laplace 2 implementation

This **axiom** program evaluates Equation 3.14 (see Section 3.1.2) using a Laplace transform.

```
Fab: (PI, PI)           -> Expression(Integer)
Rab: (PI, NNI, PI, NNI) -> Expression(Integer)
Qnl: (PI, NNI)          -> Expression(Integer)

Fab(na,nb) ==
   na2    := na**2
   nb2    := nb**2
   na4    := na**4
   result := 0::Expression(Integer)
   for la in 0..(na - 1) repeat
      for lb in 0..(nb - 1) repeat
         if (abs(la - lb) = 1) then
            rab    := Rab(na,la,nb,lb)
            result := result + max(la,lb) * rab**2
   ((nb2 - na2)/(3*nb2*na4)) * result

Rab(na,la,nb,lb) ==
   expr   := Qnl(na,la) * Qnl(nb,lb) * r**3
   result := laplace(expr,r,s)
   eval(result,s = (na + nb)/(na*nb))

Qnl(n,l) ==
   term1 := r**l
   term2 := (2/n)**((2*l + 3)/2)
   numer := factorial(n - l - 1)
   denom := 2*n * factorial(n + l)**3
   term3 := sqrt(numer/denom)
   term1 * term2 * term3 * laguerreL(2*l+1,n+l,2*r/n)
```

Appendix C

VC generation—source code

In this appendix we give the Aldor source code for various parts of the prototype lightweight verification condition generator that we have developed (see Section 5.3). The full program contains about 8500 lines of code, most of which is quite dull. Instead we have extracted parts of the program which may be of interest to the reader.

C.1 Annotating programs

The first task of our VC generator is to translate the annotations provided by the user into a form where each identifier is marked with its state. During this phase the VC generator will also add annotations to statements for which the semantics are clearly understood. The most obvious example of this is the assignment statement. The top level of the annotation phase is implemented by the `annotate!` function which modifies the internal representation of the annotated program *in situ*.

In the code below, the % symbol can be regarded as an abbreviation for the Aldor domain of annotated programs. The meaning of other domains are given in the table below:

`LslSymbol`	Identifier with a specific name and state
`LarchAldorSpecification`	User annotation
`SExpression`	LISP expression, *e.g.* pairs

```
annotate!(p:%):LslSymbol == {
   -- Perform renaming on specifications before and after the
   -- main annotation phase for this node.
   import from Character;

   local result:LslSymbol;
   local spec, fixed, final:LarchAldorSpecification;
   local modList:List(SExpression);
   local preState, postState, fixedState:List(LslSymbol);
   local specState, otherState:List(LslSymbol);


   -- Extract the user's specification and fix any missing state
   -- annotations. Unqualified identifiers in the pre-condition
   -- are defined to be in the pre-state. Unqualified identifiers
   -- in the post-condition are assumed to be in the post-state.
   spec  := getUserSpecification(p);
   fixed := fixSpec(spec, char("^"), char("'"));


   -- Rename the identifiers listed in their pre-state. In doing
   -- so we note and new identifiers which have been defined.
   (fixed, specState) := renameState(fixed, char("^"));
   setUserSpecification!(p, fixed);


   -- Check to see if the user's specification lists identifiers
   -- which are modified by the body of the expression and note
   -- their current state so that we can mutate them later.
   modList := getUserModifiesList(p);
   preState := getCurrentState(modList);


   -- Here is the main (recursive) annotation step.
   result := annotateNode!(p);


   -- Get the state of potentially mutated variables now.
   postState := getCurrentState(modList);


   -- Fix any mutations which might have occurred and
   -- update the symbol table.
   fixedState := fixMutations(preState, postState);
   for sym in fixedState repeat
      insertIntoTable!(sym);
```

```
      -- Now rename the identifiers listed in the post-state.
      spec := getUserSpecification(p);
      (final, otherState) := renameState(spec, char("'"));
      setUserSpecification!(p, final);


      -- If everything is okay then otherSyms ought to be empty. If
      -- not we tell the user (code omitted).
      -- if (not(empty?(otherState))) then { ... }


      -- Decorate this node with the list of new LSL symbols. This
      -- helps later phases of the analyser know when new logical
      -- symbols have been introduced due to state changes.
      addNewSymbols!(p, postState);
      addNewSymbols!(p, fixedState);
      addNewSymbols!(p, specState);
      addNewSymbols!(p, otherState);


      -- The return value is the logical symbol representing the
      -- value of this node (if any).
      result;
   }
```

The implementation of the recursive function `annotateNode!` is trivial:

```
   local annotateNode!(p:%):LslSymbol == {
      -- Extract the internal representation.
      local s:SourceBodyType := (rep p).srcBody;


      -- Check each case in turn: start with the easy cases.
      empty?(p)      => NoSymbol;
      unknown?(p)    => NoSymbol;


      -- Proceed with the other cases.
      (s case applyExpr)      => annotateApply!(p);
      (s case assignExpr)     => annotateAssign!(p);
      (s case declareExpr)    => annotateDeclare!(p);
      (s case idExpr)         => annotateId!(p);
      (s case sequenceExpr)   => annotateSequence!(p);


      -- Other cases we just want to ignore.
      NoSymbol;
   }
```

Finally we give an example of one of the low-level annotation functions:

```
local annotateId!(id:%):LslSymbol == {
   -->> (Id <id> <typ>)
   import from SymbolTable(String, LslSymbol);

   local sort, idsym, symb:String;
   local found:Boolean;
   local lslsym:LslSymbol;


   -- Extract the name and sort name of this identifier.
   sort := flattenType((rep id).srcBody.idExpr.type);
   idsym := ((rep id).srcBody.idExpr.sym);


   -- Combine to give a symbol name then check to see if
   -- it has an associated LSL symbol.
   symb := idsym + ":" + sort;
   (found, lslsym) := lookupInTable(symb);


   -- Did we find it? If yes then do nothing otherwise ...
   if not(found) then {
      -- This is the first time that the symbol has been
      -- declared so this must be its declaration. Create
      -- an LSL representation of this symbol and add it
      -- to the symbol table.
      lslsym := symb::LslSymbol; -- :: means coerce().
      insertIntoTable!(lslsym);


      -- Decorate this node with the new LSL symbol.
      addNewSymbols!(id, list(lslsym)$List(LslSymbol));
   }


   -- Return the LSL symbol associated with this node
   lslsym;
}
```

C.2 VC generation

The generation of VCs is straightforward once the annotation phase has completed its work. Any node which does not have a post-condition in the user-annotation is recursively verified

to obtain an estimate of the actual post-condition. Nodes which do have user-supplied post-conditions are not examined and the user annotation defines its semantics (the lightweight approach). Pre-conditions are transformed into VCs while post-conditions are used to update our knowledge of the current state of the program.

First we show the top-level verification function: the `Path` domain is used to hold the list of VCs generated for this program and the current state. Each VC is marked with the position at which it appears in the source code of the user's annotated program.

```
verify(node:%, P:Path):Path == {
   -- Deal with the preconditions (VC generation)
   local path := verifyPreState(node, P);


   -- Has the user given us a post-conditions? If so then we
   -- skip over the body of this node and trust that the post-
   -- condition tells us enough about what is going on.
   if (empty?(getUserPostCondition(node))) then
         path := verifyNode(node, path);


   -- Deal with the postconditions (update context) and
   -- return the resulting VCs and new context.
   verifyPostState(node, path);
}
```

Generation of VCs is a little more involved: we store them together until we have finished the verification phase before giving them to the user. This helps to retain modularity and allows the VCs to be converted into the object language of a theorem prover with ease.

```
local verifyPreState(node:%, P:Path):Path == {
   -- This routine deals with the preconditions.
   local items:List(VCitem);
   local vcs:Predicate;
   local vcitem:VCitem;
   local context:Context;

   local path:Path := P;
   local result:List(Path) := empty();
   local srcpos:SourceInfo == getSourcePosition(node);
```

```
    -- Extract the context and the VCs.
    context := context(path);
    items   := vcitems(path);


    -- Extend the context by any assertions
    context := conjoin(context, getUserAssertion(node));
    context := conjoin(context, getInternalAssertion(node));


    -- Add the automatically generated pre-condition to context
    context := conjoin(context, getInternalPreCondition(node));
    vcitem  := [ c == copy(context), p == empty(), pos == srcpos,
                 label == "Semantic pre-condition",
                 syms == empty() ];
    items   := cons(vcitem, items);


    -- Generate VCs from the user's pre-condition
    vcs    := getUserPreCondition(node);
    vcitem := [ c == copy(context), p == vcs, pos == srcpos,
                label == "User-supplied pre-condition",
                syms  == (rep node).srcSyms ];
    items  := cons(vcitem, items);


    -- Return the path using the new contexts.
    new(position(path), items, context);
}
```

The processing of the post-conditions is similar and will not be shown here. The function `verifyNode` is almost identical to `annotateNode!` and will not be shown either. Instead we show how assignments are analysed:

```
verifyAssign(p:%, P:Path):Path == {
    local lhs, rhs:%;


    -- Access the LHS and RHS of the assignment
    -- branch of the union has been given to us.
    lhs := (rep p).srcBody.assignExpr.decl;
    rhs := (rep p).srcBody.assignExpr.expr;


    -- Verify the RHS then the LHS.
    verify(lhs, verify(rhs, P));
}
```

Bibliography

[1] ADAMS, A., GOTTLIEBSEN, H., LINTON, S., AND MARTIN, U. VSDITLU: a verified symbolic definite integral table look-up. In *CADE 16* (1999), pp. 112–126.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers—Principles, Techniques, and Tools*. Computer Science. Addison-Wesley, 1986.

[3] BARRAS, B., BOUTIN, S., ET AL. The COQ proof assistant reference manual (Version 6.1). Tech. rep., INRIA, 1996.

[4] BARRETT, G. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering 15*, 5 (May 1989), 611–621.

[5] BAUER, A., CLARKE, E., AND ZHAO, X. Analytica—an experiment in combining theorem proving and symbolic computation. *Journal of Automated Reasoning*, 3 (1998), 295–325.

[6] BRANSDEN, B. H., AND JOACHAIN, C. J. *Introduction to Quantum Mechanics*, first ed. Longman Scientific and Technical, 1989.

[7] BRONSTEIN, M. SUM-IT: A strongly-typed embeddable computer algebra library. In *Proceedings of DISCO'96* (1996), vol. 1128 of *Lecture Notes in Computer Science*.

[8] BROOKS, F. P. *The Mythical Man-month*, anniversary ed. Addison Wesley, 1995.

[9] BROWN, R., AND TONKS, A. Calculations with simplical and cubical groups in AXIOM. *Journal of Symbolic Computation 17* (1994), 159–179.

[10] CALMET, J., AND VAN HULZEN, J. A. *Computer Algebra - Symbolic and Algebraic Computation*, first ed. No. 4 in Computing Supplementum. Springer-Verlag, 1982, ch. Computer Algebra Applications, pp. 245–258.

[11] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *Computing Surveys 17* (Dec. 1985), 471–522.

[12] CHALIN, P. On the language design and semantic foundation of LCL, a Larch/C interface specification language. Tech. Rep. CU/DCS-TR-95-12, Department of Computer Science, Concordia University, 1455 de Maisonneuve Blvd. West, Montreal, Quebec, Canada H3G 1M8, Dec. 1995.

[13] CHAR, B. W. *Maple V language Reference Manual*. Springer-Verlag, 1991.

[14] CHEON, Y., AND LEAVENS, G. T. A gentle introduction to Larch/Smalltalk specification browsers. Tech. Rep. TR 94-01, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, USA, Jan. 1994.

[15] CHETALI, B., AND LESCANNE, P. An exercise in LP: The proof of a non restoring division circuit. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 55–68.

[16] CRAIGEN, D., AND AMD TED RALSTON, S. G. An international survey of industrial applications of formal methods. Tech. rep., U.S. Department of Commerce, Mar. 1993.

[17] DAVENPORT, J. H. The AXIOM system. Tech. Rep. TR5/92 (NP2492), The Numerical Algorithms Group (NAG) Ltd., 1992.

[18] DAVENPORT, J. H., GIANNI, P., AND TRAGER, B. M. Scratchpad's view of algebra II: A categorical view of factorization. Tech. Rep. TR4/92 (NP2491), The Numerical Algorithms Group (NAG) Ltd., 1992.

[19] DAVENPORT, J. H., AND TRAGER, B. M. Scratchpad's view of algebra I: Basic commutative algebra. Tech. Rep. TR3/92 (NP2490), The Numerical Algorithms Group (NAG) Ltd., 1992.

[20] DE BRUIJN, N. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration* (1968), vol. 125 of *Lecture Notes in Mathematics*, Springer-Verlag.

[21] DECK, M. Cleanroom and object-oriented engineering: A unique synergy. In *Proceedings of the Eighth Annual Software Technology Conference* (1996).

[22] DETLEFS, D. L., LEINO, K. R. M., NELSON, G., AND SAXE, J. B. Extended Static Checking. Tech. Rep. 159, Compaq SRC, Dec. 1998.

[23] DROMEY, G. *Program Derivation*, first ed. International Computer Science Series. Addison Wesley, 1989.

[24] EVANS, D. Using specifications to check source code. Master's thesis, Department of Electrical Engineering and Computer Science, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, June 1994.

[25] EVANS, D., GUTTAG, J., HORNING, J., AND TAN, Y. M. LCLint: a tool for using specifications to check code. Pre-print, Dec. 1994.

[26] FEIT, W., AND THOMPSON, J. G. Solvability of groups of odd order. *Pacific Journal of Mathematics 13* (1963), 775–1029.

[27] FLANAGAN, D. *Java in a nutshell*, second ed. O'Reilly, 1997.

[28] FLOYD, R. W. Assigning meanings to programs. *Proceedings of the American Mathematical Society Symposium on Applied Mathematics 19* (Apr. 1967), 19–32.

[29] FOSDICK, L. D., AND OSTERWEIL, L. J. Data flow analysis in software reliability. *Computing Surveys 8 (3)* (Sept. 1976), 305–330.

[30] FRANCEZ, N. *Program Verification*. Addison-Wesley, 1992.

[31] THE GAP GROUP. *GAP–Groups, Algorithms, and Programming, Version 4*. Aachen, St Andrews, 1998. Available on the WWW at `www-gap.dcs.st-and.ac.uk/~gap`.

[32] GHEZZI, C., AND JAZAYERI, M. *Programming Language Concepts*. John Wiley and Sons, 1987.

[33] GORDON, M. J. C. *Programming language theory and its implementation*. Series in Computer Science. Prentice Hall International, 1988.

[34] GORDON, M. J. C., AND MELHAM, T. F., Eds. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993. A theorem proving environment for higher order logic, Appendix B by R. J. Boulton.

[35] GUASPARI, D., MARCEAU, C., AND POLAK, W. Formal verification of Ada programs. In *First International Workshop on Larch* (July 1992), U. Martin and J. Wing, Eds., Springer-Verlag, pp. 104–141.

[36] GUTTAG, J. V., AND HORNING, J. J. Introduction to LCL, a Larch/C interface language. Tech. Rep. TR-74, DEC SRC, July 1991.

[37] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification*, first ed. Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[38] HARRISON, J. Constructing the real numbers in HOL. *Formal Methods in System Design 5* (1994), 35–59.

[39] HARRISON, J. Floating point verification in HOL. In *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop* (Aspen Grove, Utah, Sept. 1995), W. Phillip J, T. Schubert, and J. Alves-Foss, Eds., vol. 971 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 186–199.

[40] HARRISON, J., AND THÉRY, L. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications* (UBC, Vancouver, Canada, Aug. 1993), J. J. Joyce and C. Seger, Eds., vol. 780 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 174–184.

[41] HESKETH, J., BUNDY, A., AND SMAILL, A. Using middle-out reasoning to control the synthesis of tail-recursive programs. In *Automated Deduction—CADE-11* (June 1992), D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*, pp. 310–324.

[42] HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12* (Oct. 1969), 576–583.

[43] HOMANN, K., AND CALMET, J. *Combining theorem proving and symbolic mathematical computing*, vol. 958 of *Lecture Notes in Computer Science*. Springer, 1994.

[44] HUDAK, P., JONES, S. L. P., WADLER, P., ET AL. A report on the functional language Haskell. *SIGPLAN Notices* (1992).

[45] JACKSON, D. *Aspect: A Formal Specification Language for Detecting Bugs*. PhD thesis, Laboratory for Computer Science, June MIT.

[46] JACKSON, P. Exploring abstract algebra in constructive type theory. In *Automated Deduction* (1994), A. Bundy, Ed., vol. 814 of *Lecture Notes in Artificial Intelligence*, CADE-12, Springer-Verlag.

[47] JACKSON, P. *Enhancing the NUPRL Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Apr. 1995.

[48] JENKS, R. D., AND SUTOR, R. S. *AXIOM: the scientific computation system*. Numerical Algorithms Group Ltd., 1992.

[49] JONES, C. *Logical Environments*. Cambridge University Press, 1993, ch. Completing the Rationals and Metric Spaces in LEGO.

[50] JONES, C. B. *Systematic Software Development using VDM*, second ed. Computer Science. Prentice Hall International, 1990.

[51] JONES, K. D. LM3: a Larch interface language for Modula-3, a definition and introduction. Tech. Rep. 72, SRC, Digital Equipment Corporation, Palo Alto, California, June 1991.

[52] KELSEY, T. *In preparation*. PhD thesis, University of St Andrews, 1999.

[53] KING, D. J., AND ARTHAN, R. D. Development of practical verification tools. *The ICL Systems Journal 1* (May 1996).

[54] KING, J. C. Symbolic execution and program testing. *Communications of the ACM 19(7)* (July 1976), 385–394.

[55] KNUTH, D. E. The remaining trouble spots in ALGOL 60. *Communications of the ACM 10* (Oct. 1967), 611–617.

[56] LEAVENS, G. An overview of Larch/C++: behavioral specifications for C++ modules. Tech. rep., Iowa State University, July 1997.

[57] LEAVENS, G. T. *Larch/C++ Reference Manual*. Available from the WWW at `www.cs.iastate.edu/~leavens/larchc++.html`.

[58] LEAVENS, G. T. Inheritence of interface specifications (extended abstract). Tech. Rep. TR 93-23, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011-1040, USA, Sept. 1993. Submitted to the Workshop on Interface Definition Languages.

[59] LEAVENS, G. T., AND CHEON, Y. Preliminary design of Larch/C++. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 159–184.

[60] LEAVENS, G. T., AND WING, J. M. Protection from the underspecified. Tech. Rep. CS-96-129, Carnegie Mellon University, Apr. 1996.

[61] LERNER, R. A. *Specifying Objects of Concurrent Systems*. PhD thesis, Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991.

[62] LISKOV, B., AND GUTTAG, J. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.

[63] MANNHART, N. $\prod^{\text{it}}$: an aldor library to express parallel programs. Available on the WWW at `www.inf.ethz.ch/personal/mannhart`.

[64] MARTIN, U., AND WING, J. M., Eds. *First International Workshop on Larch* (June 1992), Workshops in Computing, Springer-Verlag.

[65] MEYER, B. *Object-Oriented Software Construction*. Computer Science. Prentice Hall International, 1988.

[66] MORRISON, R. *On the Development of Algol*. PhD thesis, Department of Computational Science, University of St. Andrews, Dec. 1979.

[67] OLENDER, K. M., AND OSTERWEIL, L. J. Interprocedural static analysis of sequencing constraints. In *ACM Transactions on Software Engineering and Methodology*, vol. 1. Jan. 1992, pp. 21–52.

[68] OWRE, S., SHANKAR, N., AND RUSHBY, J. M. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993.

[69] PERRY, D. E. The Inscape environment. In *The 11th International Conference on Software Engineering* (May 1989).

[70] POLL, E., AND THOMPSON, S. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. Technical Report 6-98, Computing Laboratory, University of Kent, May 1998.

[71] POLL, E., AND THOMPSON, S. The Type System of Aldor. Tech. Rep. 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK, July 1999.

[72] POTTER, B., SINCLAIR, J., AND TILL, D. *An introduction to formal specification and Z*. Prentice Hall International, 1991.

[73] RIDEAU, P. *Computer algebra and mechanics: the JAMES software*. Wiley, 1993, pp. 143–158.

[74] ROSENBLUM, D. S. A practical approach to programming with assertions. *IEEE transactions on software engineering 21* (Jan. 1995), 19–31.

[75] SANNELLA, D. Formal program development in Extended ML for the working programmer. In *Proceedings of the 3rd BCS/FACS Workshop on Refinement* (1990), Springer Workshops in Computing, pp. 99–130.

[76] SCOTT, E. A., AND NORRIE, K. J. Using LP to study the language PL_0^+. In *First International Workshop on Larch* (June 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 227–245.

[77] SIVAPRASAD, G. S. Larch/CORBA: Specifying the behaviour of CORBA-IDL interfaces. Master's thesis, Computer Science, Iowa State University, Ames, Iowa, 50011-1040, USA, Nov. 1995. See: `www.cs.iastate.edu/`.

[78] SOMMERVILLE, I. *Software Engineering*, fourth ed. Addison-Wesley, 1992.

[79] SUN MICROSYSTEMS INC. *Programmers Overview, Utilities and Libraries*. 2550 Garcia Avenue, Mountain View, CA 94043. Chapter 6, `lint`—a Program Verifier for C.

[80] TAN, Y. M. Semantic analysis of Larch interface specifications. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 246–261.

[81] VAN HULZEN, J. A., AND CALMET, J. *Computer Algebra - Symbolic and Algebraic Computation*, first ed. No. 4 in Computing Supplementum. Springer-Verlag, 1982, ch. Computer Algebra Systems, pp. 221–243.

[82] VANDEVOORDE, M. T. *Exploiting Specifications To Improve Program Performance*. PhD thesis, Laboratory for Computer Science, MIT, 545 Technology Square, Cambridge, Massachusetts, Feb. 1994. MIT/LCS/TR-598.

[83] WATT, S. M., BROADBERY, P. A., DOOLEY, S. S., IGLIO, P., MORRISON, S. C., STEINBACH, J. M., AND SUTOR, R. S. *AXIOM Library Compiler User Guide*, first ed. NAG Ltd., Mar. 1995. Reprinted with corrections from November 1994.

[84] WING, J. M. A two-tiered approach to specifying programs. Tech. Rep. LCS/TR–299, Laboratory for Computer Science, MIT, May 1983.

[85] WING, J. M., ROLLINS, E., AND ZAREMSKI, A. M. Thoughts on a Larch/ML and a new application for TP. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Workshops in Computing, Springer-Verlag, pp. 297–312.

[86] WOLFRAM, S. *Mathematica: A system for doing mathematics by computer*, 2 ed. Addison Wesley, 1991.

[87] WYLIE, C. R. *Advanced Engineering Mathematics*, International Student ed. McGraw-Hill, 1975.

[88] ZAREMSKI, A. M., AND WING, J. M. Specification matching of software components. *Proceedings of 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Oct. 1995).