Lightweight Formal Methods For Computer Algebra Systems

Martin Dunstan Tom Kelsey Steve Linton Ursula Martin

Division of Computer Science
University of St Andrews
North Haugh, St Andrews, UK
{mnd, tom, sal, um}@dcs.st-and.ac.uk
http://www-theory.dcs.st-and.ac.uk

Abstract

In this paper we demonstrate the use of formal methods tools to provide a semantics for the type hierarchy of the AXIOM computer algebra system, and a methodology for Aldor program analysis and verification. We give examples of abstract specifications of AXIOM primitives, and provide an interface between these abstractions and Aldor code.

1 Introduction

We describe work in progress at St Andrews to apply formal methods and machine assisted theorem proving techniques to improve the robustness and reliability of computer algebra systems. This project considers the use of the Larch [7] approach to formal methods through specifications and uses AXIOM [8] for the computer algebra system. We do not exclude other formal methods systems such as VDM [9] or Z [13] nor do we exclude applications to other computer algebra systems (CAS) such as Mathematica [17] or Maple [14]. Indeed the weaker type systems used by the latter packages may benefit more from our approach than AXIOM can.

In the remainder of this introduction we motivate the project, provide an overview of the structure of AXIOM and its extension language, Aldor, describe our approach to formal methods to CAS, introduce the formal methods tools we have used, and provide a methodology for applying the tools to a computer algebra environment. The second section consists of an overview of our abstract specifications, together with worked examples from the AXIOM algebra type-hierarchy. This is followed by a section describing the use of Larch/Aldor specifications (which provide an interface between the abstractions of Section 2 and Aldor) and the use of forwards program analysis for the generation of verification conditions. The final section is a brief statement of conclusions formed and issues arising.

1.1 Motivation

CAS are large and complicated software packages. It is not necessary for users to understand the whole system, and developers may be unsure as to whether to trust an existing procedure in a new context or produce a new implementation from scratch. Moreover, while computer algebra algorithms are generally sound, there can be hidden dependencies and implicit side conditions present which can lead to erroneous or misinterpreted results. Examples include inconsistent choice of branch cuts in integration algorithms [4], invalid assumptions for the types of arguments of a function or poorly documented side-conditions. An example of incorrect typing is the AXIOM '`SingleInteger`' type. This type represents finite (machine) precision integers which wrap around at a machine dependent value. However, '`SingleInteger`' has type '`EntireRing`' and hence has no non-zero zero divisors. This is incorrect since '`(2**16:SINT)**2 = 0`' in AXIOM on a 32 bit machine. However, the implemented version is more useful for most practical purposes.

1.2 AXIOM and Aldor

AXIOM [8] (originally ScratchPad 2 from IBM [3]) is a strongly typed CAS with a two-level object model: every value has a type which is used to prevent values being used in inappropriate situations. Computation is carried out at the AXIOM *domain* level: domains provide an environment in which several functions can act on elements of the same type. The type of a domain is an AXIOM *category*. For example the functor '`Matrix`' takes a domain of type '`Ring`' and returns a domain of type '`MatrixCategory`'. '`Ring`' provides algebraic operations for matrix elements, '`MatrixCategory`' provides linear algebra operations for matrices. A category defines the typing restrictions of functions in a domain environment. Categories can be created by Aldor users and developers, but the inbuilt hierarchies of algebraic and data structure categories are sufficient for most purposes.

Like other CAS, AXIOM can be used interactively via an interpreted language but there is also a compilable extension called Aldor (formerly known as Axiom-XL [16] and $A^\sharp$ [15]). Aldor is a strongly-typed, imperative language with a number of features more commonly found in the functional programming world such as higher-order functions. As in AXIOM itself, Aldor has a two-level object model with inheritance which gives it the flexibility and power necessary for implementing and organising computer algebra routines. It should be noted that the two-level object model is unusual in both CAS and general purpose programming languages.

The top level of the object model is the category which can be regarded as the defining a fragment of an abstract

data-type (ADT) interface. The implementation of an ADT forms the bottom level of the model and is called a domain; the interface of a domain can be constructed by referring to different categories. For example '`MatrixCategory`' defines row and column structures in terms of the category '`FiniteLinearAggregate`' contained in the AXIOM data structure hierarchy.

In addition to providing information about the exported values of a domain, an Aldor category allows domain-manipulating routines to provide restrictions the kinds of domains they will accept and make promises about the type of domain it will return.

Types and functions can be used as (constant) values in expressions to provide parametric polymorphism and currying while generators (sometimes known as co-routines) give the programmer an efficient and generic method of iterating over members of aggregate data types. The language also provides *post facto* extensions which allow the behaviour of existing types and functions to be extended.

The current Aldor compiler is able to produce libraries for AXIOM as well as LISP, C and stand-alone executables for the native platform of the user. Thus Aldor can be used as a general purpose programming language in addition to implementing computer algebra algorithms.

1.3 Formal Methods and CAS

There are a number of ways that CAS and theorem provers can be linked together; the approach with which we are concerned in this paper is the use of formal methods for software development to improve the robustness and reliability of computer algebra systems. More specifically we are investigating how the Larch [7] approach to formal specification and program design applies to a CAS such as AXIOM.

During software development there are a number places where errors can be introduced. These can be broadly categorised as the design, implementation and maintenance phases. Since many computer algebra routines have been studied and reviewed over many years we can be confident that they have been well designed and that the majority work as their authors intended them to. However, we believe that it is more likely that errors will occur through the inappropriate use of these routines, either by invoking them with invalid arguments or applying them in the wrong situations.

In this paper we suggest that developers should consider specification and formal methods during software development. The much-cited advantages include improving understanding of the problem during the design phase, fewer design errors since each stage must have a mathematical justification and fewer bugs in the final implementation. In addition, machine checking and proof assistants can help provide proofs of these justifications, as well as helping the specifier check that the specifications have their intended properties. A program which satisfies a faulty specification is no safer than one which fails to satisfy a correct one.

Having obtained an implementation there is still a chance that there will be bugs in it. These range from simple typing mistakes through to errors in the program logic: the design process may have produced a specification which is easily implementable but this does not stop the programmer making mistakes!

To help with these kinds of problems the programmer will often turn to mechanical program checkers such as the popular `lint` program for C. This program is able to analyse sections of source code and identify some mistakes which may be easily missed by the programmer. There is a limit to how much work these analysers can perform since they do not take into account the objectives of the program. By allowing the static analyser to access the specifications of the program it may be able to detect more classes of mistakes [5]; the eventual goal is to show that the program completely satisfies the specification.

Strongly typed systems such as AXIOM avoid many of these problems since the type checking of the interpreter or the compiler can enforce restrictions on argument types (e.g. an implementation of the factorial function over the non-negative integers can be defined to only accept values of this type: weaker systems must use an integer type and perform runtime argument checking). However, a strong type system does not eliminate this problem since it may be impractical to be overly strict in the choice of types: having floating point types `FloatRoundToInfinity`, `FloatRoundToZero` etc. to express different systems of rounding modes is excessive and severely complicates a given implementation.

The solution we advocate is to clearly specify the intent of a given function or subroutine in a machine checkable format and use appropriate formal methods tools to highlight possible violations. Such violations may indicate the presence of bugs which can be removed before they become expensive to eliminate. The specifications will also help developers by providing clear, concise and (hopefully) unambiguous documentation.

1.4 Larch

When choosing a specification language one must decide what level of abstraction is required. If the specification language allows only abstract (programming language independent) specifications then it may be difficult or even impossible to obtain an implementation from them. Alternatively a specification language which is able to cope with the features of different implementation languages, such as inheritance, pointers and procedures with side-effects, is likely to be too complex to be used with confidence.

The Larch [7] family of languages and tools tackles this problem by adopting a two-tiered specification system. At the top level specifications are written in an algebraic specification language called the Larch Shared Language (LSL) which is based on first order logic with equality and induction. The second level is a behavioural interface specification language (BISL) which is used to specify the details of a particular implementation such as procedure side-effects and aliasing using primitives defined in LSL. The tiered system gives Larch great flexibility since each target programming language uses a specifically designed BISL.

At the time of writing there are at least 14 Larch BISLs. Larch/CLU was the first Larch language and formed the basis for subsequent Larch languages, each one looking at different features of the target programming language. For example, Larch/C++ [12] has been used for investigating inheritance in BISL specifications [11] and Larch/Modula-3 [10] for concurrency issues.

The basic unit of specification in LSL is a *trait*. A trait introduces operator names, signatures (involving type, or *sort*, names), and a set of axioms which define properties of the operators. Traits can be combined incrementally to produce structured specifications. It is also possible to use the same trait in several different contexts by renaming sorts

and by renaming or overloading operators. An LSL specification provides the BISL with reference points for describing a mapping from the types in the interface specification to the sorts in the associated trait.

The Larch Prover (LP) is an automated proof assistant based on classical set theory. LP can orient the properties of an LSL trait into induction and deduction rules, AC operator theories and rewrite rules. The user can then prove assertions about semantic properties of the trait. LP proof tactics include proofs by induction, contradiction and cases, as well as standard automated reasoning techniques such as normalisation and critical-pair computation.

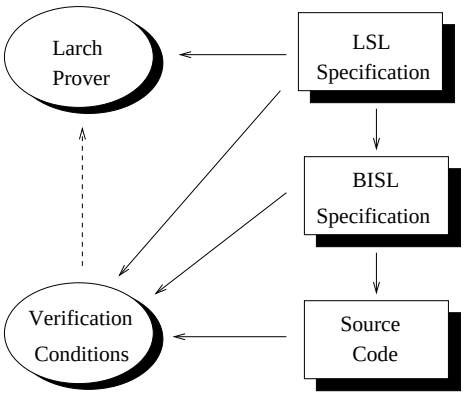

Figure 1: Larch/Aldor Development Diagram

Figure 1.4 gives an overview of how software is developed under the Larch/Aldor system described in this paper. The BISL specification (see Section 1.5) uses the primitives defined by LSL specifications (Section 2) to document the source code. These specifications are used to generate verification conditions (Section 1.7) from the source code; the user may attempt to discharge these conditions using a theorem proving system such as LP.

1.5 Larch BISLs

Although each Larch BISL is tailored to the syntax of the target programming language there are similarities between all of them. The basic structure of a BISL specification comprises of pre- and post-conditions (usually introduced by the keywords `requires` and `ensures`) for the function, and a list of client-visible state potentially modified by the function (a `modifies` clause). Some Larch BISLs permit the use of multiple pre- and post-conditions as in the Larch/C++ specification of a square root function

```
double squareRoot(double x) throw(NEG_ROOT)
{
   requires (x >= 0);
   ensures  (result = sqrt(x));

   requires (x < 0);
   ensures thrown(NEG_ROOT) = 1;
}
```

where `sqrt()` is an LSL operator which would be defined in a separate trait. Omitting a `modifies` clause is equivalent to writing '`modifies nothing`'.

BISL specifications refer to abstract objects specified in LSL: in the example above the identifier '`x`' represents the value of the abstract LSL object corresponding to the C++ function parameter '`x`'. The symbol '`result`' is a specification variable representing the return value of the function.

The general aim when writing Larch specifications is to keep the BISL specification as simple as possible: do as much work as possible in LSL, and only put the language/implementation specific details in the BISL tier. The justification for this is that LSL specifications are easier to reuse, have a straightforward semantics and are easier to check for errors and omissions.

A BISL specification should only be concerned with program state and is therefore only suited for the imperative parts of programming languages. Succinctness is acheived through the use of operators provided by an LSL theory and the resulting specification is hopefully clearer and more precise than pure prose. These specifications can, therefore, be used both as documentation to the user and as a guide or contract to the implementor.

1.5.1 Work required to develop a Larch BISL

Developing a new Larch interface language is by no means a simple task and one must decide how much work needs to be tackled to meet the needs of the users of the language. At one end of the spectrum the language might conceivably consist of just a syntax definition providing users with nothing more than a clear and concise form of program documentation. This approach is only useful if the specifications being created are not too complex and the meaning of any operators used is clear.

At the other end of the spectrum we need a formal model of the semantics of the target programming language as well as a model of computation suitable for the Larch BISL [2]; we also need to know the semantics of the BISL. Once this background theory has been obtained (a non-trivial problem) we turn to tool support for developers using the BISL.

A syntax checker might be considered to be the bare minimum that any Larch system ought to provide, while an interactive program development and verification environment is perhaps an ultimate tool (e.g. Penelope for Larch/Ada [6]). In between we have static data-flow analysers which are able to perform `lint`-like checks augmented by the information provided in specifications (e.g. LcLint [5]) and automated verification condition generators.

1.6 Detecting Bugs

A tool such as LcLint uses the Larch interface specifications to enable it to perform checks that standard `lint` can not. For example, the interface specification of a function can state what part of the client-visible state might be modified by the function. Although this has a difficult proof obligation in general, LcLint can highlight problems which might otherwise go undetected. LcLint can also be particularly useful if the programmer is adopting an abstract datatype style of programming in C and wishes to ensure that they have not accidentally violated their abstraction barriers.

1.7 Program Verification

Throughout this paper we refer to program verification and verification condition generators for imperative programming languages. A verification condition generator takes a

program and its specification as input and reduces them to a set of logical statements (called verification conditions). If these statements can be shown to be true then the program is considered to be correct with respect to its specification. However, proving that an imperative program is correct with respect to a given specification is undecidable in general.

The generation of verification conditions requires a formal semantic model of the implementation language which describes how each statement and control structure of the program alters the runtime state. These models are complex for commonly used imperative programming languages such as C or Ada and so the programmer may decide to reimplement the program in a language which has a simpler semantic model and is therefore more amenable to mathematical analysis. Functional programming languages such as ML and Haskell are examples of this. Alternatively the program may be implemented using a subset of features from the target programming language where a formal model of the semantics of the subset is clearly understood.

In this paper we propose a different method: use the Larch/Aldor interface specifications as high-level operational semantics. Assuming that we can provide formal semantics for a small set of Aldor features such as assignment and function application, a lightweight program verifier may be written. This program will examine Aldor programs annotated with Larch/Aldor specifications and inform the user of any verification conditions that it is unable to discharge.

It is our intention that the verification conditions generated by our tool will be given to the user to be discharged by hand or with the assistance of a theorem prover or proof assistant. Indeed the user may simply wish to note any interesting verification conditions and continue working with the assumption that they are true.

Our proposal is intended to reduce the amount of work required to implement a program verification tool for Aldor programs but relies on the interface specifications being sound. These specifications must also provide sufficient information to assist the discharging of the verification conditions produced but we do not insist that they are complete.

2 Specifying the Axiom Library in LSL

In this section we discuss LSL specifications of AXIOM categories, and provide examples related to the case study contained in Section 3.

We restrict our discussion to the algebraic hierarchy supplied with AXIOM release 2.1, with particular reference to the set of categories shown in Figure 2. The arrows in this figure represent inheritance of operational structure.

The basic category for describing collections of objects is '`SetCategory`'. A descendent of '`SetCategory`', for example '`SemiGroup`', is defined within AXIOM in terms of

1. a set of documented axioms that are expected to apply to elements of any domain of type '`SemiGroup`'
2. a set of conditional attributes that a domain of type '`SemiGroup`' may or may not have;
3. a set of operation names, each having a signature possibly involving other categories;
4. a set of methods for implementing some or all of the operations;
5. the associated axioms, attributes, operations and methods of any ancestor categories.

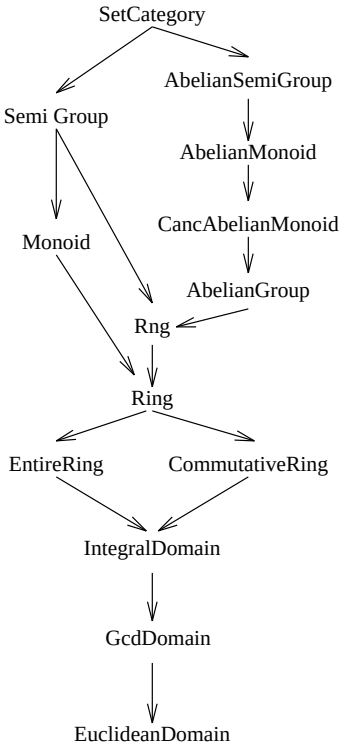

Figure 2: A sub-dag of the AXIOM algebra hierarchy

For example, the category '`SemiGroup`' provides the category '`SetCategory`' with a closed multiplicative operator which by axiom is associative and by conditional attribute may be commutative.

The strong typing of AXIOM objects stems from the signatures mentioned above: the system will reject commands or procedures which violate operation signatures. It is however possible for system users and developers to violate the axioms and conditional attributes (as seen in the '`SINT`' example). Moreover the enforcement of type-correctness does not guarantee that categories preserve the precise mathematical semantics of the structures they are intended to represent. Both these potential shortcomings are addressed by providing LSL specifications for each category in the hierarchy. By providing a formal semantics for the axioms, attributes and inherited properties of each category we allow proofs that required properties hold throughout the hierarchy of specifications. By providing unambiguous mathematical descriptions of (and restrictions on) primitives defined within categories we obtain reference points for proofs of procedure correctness at the interface level.

2.1 '`EuclideanDomain`'

The LSL specification of '`EuclideanDomain`' (Figure 3) inherits the properties of the specification of '`GcdDomain`', and includes properties of traits which represent the range sorts of '`euclidSize`' and '`divide`'. When read by the Larch Prover these inclusions provide 123 facts which can be used to check that expected properties have been inherited. Names and signatures are introduced for three AXIOM operators and two conditional attributes. The assertions pro-

```
EuclidDomainCat (ED) : trait
includes
   GcdDomainCat (ED for GD),
   EDRecord,
   EnumerableTO
introduces
   euclidSize : ED → N
   isSizeLess : ED,ED → Bool
   divide : ED,ED → R
   multiplicativeValuation : → Bool
   additiveValuation : → Bool
asserts
   ∀ x,y,mult,rem : ED, r1 : R, n,m : N
     (x ¬= 0 ∧ y ¬= 0) ⇒
       ¬(euclidSize(x*y) < euclidSize(x));
     (y¬=0 ) ⇒ ∃ mult (∃ rem ((x=(mult*y)+rem)
       ∧ (rem = 0:ED ∨ (rem ¬= 0:ED
       ∧ (euclidSize(rem) < euclidSize(y))))));
     isSizeLess(x,y) ==  ¬(y:ED = 0)
        ∧ (x=0 ∨ euclidSize(x) < euclidSize(y));
     euclidSize(divide(x,y).remainder)<euclidSize(y);
     multiplicativeValuation ⇒
      euclidSize(x*y) = euclidSize(x)*euclidSize(y);
     additiveValuation ⇒
      euclidSize(x*y) = euclidSize(x)+euclidSize(y)
implies
   ∀ x,y: ED, n,m : N
     (x ¬= 0 ∧ y ¬= 0 ∧ ¬(isUnit(y))) ⇒
        euclidSize(x) < euclidSize(x*y);
     (x ¬= 0 ∧ y ¬= 0) ⇒ x*y ¬= 0
```

Figure 3: LSL specification of `EuclideanDomain`

```
resume by =>                          ACCEPT PREMISE
 instantiate x by xc, y by yc in EuclidDomainCat.1
  REPLACE VARIABLES BY CONSTANTS
 res by cases euclidSize(xc) = (euclidSize(xc*yc))
  CASE SPLIT
   instantiate  x by xc, y by xc*yc in EuclidDomainCat.2
    prove ~(xc * yc = 0)      INTERMEDIATE RESULT
    instantiate x by xc, y by yc in EuclidDomainCatTheorem.2
    [] conjecture   INTERMEDIATE RESULT HAS BEEN PROVED
   declare operator skolmult : -> ED    SKOLEMIZATION
   fix mult as skolmult in EuclidDomainCatTheorem.2.1
    ELIMINATE EXISTENTIAL OPERATOR
   declare operator skolrem : -> ED
   fix rem as skolrem in EuclidDomainCatTheorem.4
    res by cases skolrem = 0
     instantiate x by xc, y by skolmult*yc in RingCat.10
      USE UNIQUENESS OF UNITY IN A RING
     [] case skolrem = 0
      prove skolrem = xc - (skolmult * xc * yc)
       resume by contradiction   SUPPOSE NOT
          complete   COMPUTE CRITICAL-PAIR EQUATIONS
         [] contradiction subgoal
       [] conjecture
     inst x by xc, y by (1-(skolmult*yc)) in EuclidDomainCat.1
     complete
     [] case ~(skolrem = 0)
   [] case euclidSize(xc) = euclidSize(xc*yc) 1ST CASE PROVED
    inst x by euclidSize(xc), y by euclidSize(xc*yc) in STO.1
   [] case   2ND CASE PROVED BY THE STRICT TOTAL ORDER AXIOM
 [] => subgoal
[] conjecture              THE RESULT HAS BEEN PROVED
```

Figure 4: LP proof of `EuclideanDomain` implication

vide formulae relating variables to operators. The first two assertions are the standard axioms for a valuation function from $D \setminus \{0\}$ into $I\!N$, where D is a domain. The remaining assertions are formalisations of comments contained in the AXIOM documentation of '`EuclideanDomain`'.

The implications in Figure 3 are facts in the first order theory generated by the specification. The first is a standard result taken from a textbook [1] which is true for every abstract Euclidean ring. Writing δ for '`euclidSize`' we have: **Proof of 1st Theorem:** $\delta(x) \leq \delta(xy)$ by assertion. If $\delta(x) = \delta(xy)$, then $x = mxy + r$ for some m, r in the domain, with $r = 0$ or $\delta(r) < \delta(xy)$. If $r = 0$, then $x = mxy$ and so $my = 1$, whence y is a unit. Conversely $\delta(x) \leq \delta(x(1 - my)) = \delta(r) < \delta(xy) = \delta(x)$, giving another contradiction.

The LP proof given as Figure 4 follows this proof closely, although the existential operators binding the multiplier and remainder have to be eliminated by Skolemization. The second implication is a straightforward consequence of the no (non-zero) zero divisors axiom, which is inherited from '`EntireRing`'. Proving results in the theory of a trait serves three purposes:

1. we increase our confidence that the properties of included traits have combined with the assertions to produce a specification of the required computer algebra structure.

2. we show that the LSL theory of the trait is contained within the classical theory of (in this case) Euclidean rings.

3. specifications which include the trait will have the implications available as theorems, making it easier to prove properties of more complicated structures. For example the Larch Prover proof of the first implication in Figure 3 uses the uniqueness of the multiplicative identity, which was proved as an implication in the LSL specification of '`Ring`'.

Another way of considering the implications is that failure to prove expected results indicates that the trait does not specify the intended object. A large proportion of time spent de-bugging LSL specifications consists of the identification of points where such proofs fail, and correcting the specification accordingly.

2.2 Integers in AXIOM

We now provide a specification of AXIOM's model for the integers, the category '`IntegerNumberSystem`'. For ease of exposition the specification (Figure 5) has been simplified so that certain AXIOM operations (such as modular arithmetic) have been omitted. The inclusions show '`IntegerNumberSystem`' is a totally-ordered Euclidean domain of characteristic zero. We introduce operators which allow conversion to the range space for '`euclidSize`', iteration over the sort in the order $0, 1, -1, 2, -2, 3, \cdots$, and provision of a maximum value from a pair of values. The final two assertions ensure that domain operations are compatible with the ordering.

The attribute '`multiplicativeValuation`' requires that the '`euclidSize`' is a homomorphism from the sort '`INS`' into $I\!N$. The attribute '`canonicalUnitNormal`' requires that associates have the same canonical normal form. The '`generated by`' clause allows induction over the sort, with '`init`' as basis value. This induction schema, together with a case split $x < 0, x = 0, 0 < x$, is used to obtain an LP proof of the implied result. Proving the result shows that

```
IntNumberSystCat (INS) : trait
includes
   EuclidDomainCat (INS for ED),
   StrictTotalOrder (<, INS),
   CharZeroCat (INS)
introduces
   __*__ : INS,N → N
   convert : INS → N
   init : → INS
   nextItem , abs: INS → INS
   max : INS,INS → INS
asserts
   INS generated by init, nextItem
   ∀ x,y : INS, n : N, u:UF
      multiplicativeValuation;
      canonicalUnitNormal;
      init == 0;
      nextItem(x) = nextItem(y) == x = y;
      nextItem(x) ¬= init;
      nextItem(x) == if x = 0 then 1 else
                     if x < 0 then 1-x else -x;
      max(x,y) == if ¬(y < x) then y else x;
      convert(x) = (max(x))*(succ(0):N);
      x ¬= 0 ⇒ euclidSize(x) = convert(x);
      (0 < x) ∧ (y < z) ⇒ (x*y) < (x*z);
      (x < y) ⇒ (x+z) < (y+z)
implies
      converts
        euclidSize
      exempting euclidSize(0)
```

Figure 5: LSL specification of `IntegerNumberSystem`

when the interpretation of all other operators is fixed, there is a unique interpretation for '`euclidSize`', apart from the value '`euclidSize(0)`' which exists, but is unspecified.

We now have everything in place for specifying the AXIOM domain '`Integer`' (Figure 6). The attribute '`canonicalsClosed`' ensures that a product of canonicals is itself canonical. '`unitNormal(x)`' was first introduced in the specification of '`IntegralDomain`', and is required to return a record '`[unit, canonical, associate]`' such that '`x = unit*canonical`' and '`associate*unit = 1`'. We have now interpreted this requirement in the knowledge (proved implications) that for this particular integral domain the units are 1 and -1, and the associate class of '`x`' consists of '`x`' and '`-x`'.

```
AXInteger : trait
includes
   IntNumberSystCat (Z for INS)
introduces
   factorial : Z → Z
asserts
   ∀ x,y : Z
      canonicalsClosed;
      unitNormal(x) == if x < 0 then [-1,-x,-1]
                       else [1,x,1];
      factorial(0) == 1;
      0 < x ⇒ factorial(x) = x*factorial(x-1)
implies
    ∀ x,y : Z
      (x =1 ∨  x = -1) ⇔ isUnit(x);
      areAssociates(x,y) ⇔ y = -x ∨ y = x;
```

Figure 6: LSL specification of `Integer`

Other AXIOM domains of type '`EuclideanDomain`' can be specified in a similar manner. A system developer need only check that a new domain specification is consistent with the definitions of the operators introduced in the traits associated with the AXIOM algebraic hierarchy. The LSL traits provide the abstract concepts required for reasoning about AXIOM and Aldor programs. In particular formal definitions of the sort '`Z`' and the operator '`factorial`' have been given; these will be used as reference points at the interface level, as discussed in the following section. The LP proofs of results in the theories of the traits give the developer confidence that the abstract concepts are flexible enough to allow many implementations, but rigourous enough to rule out mathematically incorrect results.

3 Larch/Aldor

In this section we look briefly at the design and implementation of a Larch behavioural interface specification language (BISL) for Aldor. We then concentrate on showing how this language and associated tools can be used to assist in the development of Aldor programs, either for stand-alone applications or for AXIOM library functions.

As described in the introduction we feel that full program verification is perhaps excessive for the types of systems we are dealing with. Instead we adopt a more lightweight approach using tools to generate verification conditions based on Larch interface specifications. The user may wish to note the verification conditions as interesting facts to be recorded in the documentation, or try to discharge them using their favourite theorem proving system or by hand.

3.1 Program Analysis

Initially a simple data-flow analyser for Aldor was implemented in Aldor itself. This program accepts the parse tree from the current Aldor compiler as input and emits warning messages if certain data-flow anomalies are detected. These include use-before-definition and definition-without-use anomalies. The compiler already detects these anomalies in certain circumstances but our analyser attempts to be more thorough. For example, in the Aldor program

```
local a,b,c:Integer;

a := randomValue();
b := randomValue();

if (a > b) then
   c := 42;

print << c << newline;
```

the variable '`c`' is only defined if '`(a > b)`'. Our analyser follows both possible execution paths and warn the user that '`c`' *might* be undefined at the last line.

This initial work has given the authors valuable experience for integrating new components into the existing Aldor compiler. Using this knowledge we are proceeding to implement a more powerful tool to provide "lightweight" verification of Aldor programs written using Larch/Aldor.

3.2 Program Verification

If a function F has pre-condition $\{P\}$ and post-condition $\{Q\}$ then an application of F will at least generate a verifica-

tion condition that $\{P\}$ is satisfied according to the current context (program state). Our verification condition generator will assume that $\{Q\}$ specifies the semantics of F and use it to extend the context. We trust that $\{Q\}$ contains sufficient information to help discharge subsequent verification conditions: if not the user may need to extend $\{Q\}$.

3.2.1 Loop-free programs with assignment

Consider the following Larch/Aldor program fragment:

```
Factorial: (n:Integer) -> Integer;
   ++} uses     AXInteger(Integer for Z);
   ++} requires ~(n < 0);
   ++} ensures  result = factorial(n);

a := Factorial 6;
```

where the '++}' symbol marks the Larch/Aldor specification and the other two lines are Aldor source code.

The first line of the specification states that our LSL theory can be found in the trait `AXInteger` and that the Aldor type `Integer` corresponds to the LSL sort `Z` (see Section 2.2). The `requires` clause uses the LSL '`<`' operator to compare the LSL value of the Aldor identifier '`n`' with the LSL value of the Aldor value `0`. Similarly the `ensures` line states that the result of the '`Factorial`' function (represented by the specification variable '`result`') is equal to the value obtained by applying the LSL '`factorial`' operator to the LSL value of the Aldor identifier '`n`'.

As described earlier, the `requires` clause defines the pre-condition of this function while `ensures` defines the post-condition. If the pre-condition holds when '`Factorial`' is invoked then our specification states that it will terminate and when it does the post-condition will hold. If the pre-condition is not satisfied then the behaviour of the function is undefined: it might run for ever, return any value *etc.*

To allow useful verification conditions to be generated we require a specification of the assignment operator. This cannot be defined in Larch/Aldor and would be part of the meta theory used by the verification condition generator. For this example we will assume its specification is

```
(:=) : (var:Variable, value:Integer) -> ();
   ++} requires declared(var);
   ++} ensures  var' = value;
   ++} modifies var;
```

The pre-condition is that the variable being assigned is declared and has storage allocated to it while the post-condition is that the variable will have the specified value. The primed notation used in the post-condition represents the value of '`var`' in the post-state to differentiate it from its value in the pre-state.

To verify that the program fragment is correct with respect to these specifications we derive pre- and post-conditions for each statement With our approach this is simply a matter of using the interface specifications with appropriate renaming:

```
a := Factorial 6;
   ++> requires declared(a) /\ (~(6 < 0));
   ++> ensures  (a' = factorial 6);
   ++> modifies a;
```

After simplification we obtain the verification condition that '`declared(a)`' must hold before the assignment is executed. Assuming this is true we can proceed to analyse the next program statement knowing that the '`a`' now has the value 6!. If our interface specifications are correct then we can be confident that our program is correct once all the verification conditions have been successfully discharged.

3.2.2 Going loopy

The specification and verification of loops has not been addressed very much in the Larch literature to date, perhaps because BISLs are mainly concerned with procedures and functions. In our approach we adopt the abstraction mechanisms of procedures and treat a loop as a building block, just like a procedure, which has pre- and post-conditions. Following the Larch approach we use the `modifies` clause to indicate which parts of the client-visible state might be altered when the loop body is executed.

In addition we allow a loop invariant to be specified along with a measure function to enable verification conditions to be generated for termination proof attempts. The measure function must be defined over a well-founded ordered set such as the natural numbers and must decrease monotonically with each iteration of the loop. It will adopt a minimum value when the loop terminates, e.g. 0 for the naturals.

When a loop is encountered a verification condition is generated to show that the pre-condition of the loop is satisfied. With our approach we do not need to immediately generate verification conditions for the loop body to obtain the state of the program when the loop terminates. Instead we rely on the post-condition to extend the current context and to provide enough information to enable the verification condition generator to proceed successfully with the statements following the loop. The loop itself can be analysed separately using the context which existed prior to the loop.

A possible implementation of the factorial function might look like the following:

```
Factorial(n:Integer):Integer ==
{
   local i, x:Integer;
   i := 0; x := 1;

   while (i < n) repeat
   ++} modifies  i, x;
   ++} ensures  (x = factorial n) /\ (i = n);
   ++} invariant x = factorial i;
   ++} measure   n - i;
   {
      i := i + 1;
      x := x * i;
   }
   return x;
}
```

There are no restrictions on the program state before the loop begins and the post-condition is that '`x`' holds the value `n`!. The only modifications to client-visible state during the execution of the loop are the variables '`x`' and '`i`' and their values after the loop are specified in the post-condition.

Applying our lightweight verification process to this routine, we begin with the pre-condition of '`Factorial`', namely $\neg(\texttt{n} < 0)$. Proceeding informally, the assignments to '`i`' and '`x`' produce the context $\neg(\texttt{n} < 0) \wedge (\texttt{x}_0 = 1) \wedge (\texttt{i}_0 = 0)$

where the subscripts are used to distinguish between the LSL values of 'i' and 'x' in different program states.

The precondition of the loop is vacuous and so the context after the loop terminates is $\neg(\mathtt{n} < 0) \wedge (\mathtt{x}_0 = 1) \wedge (\mathtt{i}_0 = 0) \wedge (\mathtt{x}_1 = \mathtt{factorial\,n}) \wedge (\mathtt{i}_1 = \mathtt{n})$. The post-condition of the function can be discharged by observing that the LSL value of 'x' in the return statement is $\mathtt{x}_1$ which is equivalent to `factorial n`.

Assuming the implementation of the loop body is correct then we can be confident that the procedure as a whole is correct with respect to its interface specification. Using induction we can show that for the base case where $(\mathtt{n} = 0)$ the loop invariant is satisfied; assuming the invariant is true when $(\mathtt{i} = k)$ then we must show that it is also true after one more iteration. This is achieved fairly easily using the axiom $(n+1)! = (n+1)n!$ which forms part of the LSL tier.

4 Issues and Conclusions

The approach of verification condition generation described in this paper is modular: we did not have to verify that the factorial function was correct to be able to generate the necessary verification conditions for '`a := Factorial 6`'. Indeed during the early stages of program development the factorial function may only exist as a procedure stub with an interface specification and no code. Similarly for loops and other block structures. Futhermore we allow the user to decide what to do with the verification conditions generated. They may attempt to discharge them (by hand or using appropriate machinery) or simply note them as useful conditions on the use of the program.

At some point in the verification process the user ought to check that the functions do indeed satisfy their interface specification but this does not need not to be repeated whenever the rest of the program is verified. This has a significant advantage when verifying large programs which are constantly evolving. Moreover the modularity of LSL specifications allows the reuse of abstractions in a variety of system-building contexts: it is sufficient to provide one trait which defines, say, strict total ordering that can be used as often as needed.

Another important consideration is that we adopt a forwards rather than a backwards analysis of the program. Forward program analysis program is generally considered to be undesirable since the verification condition generator will reach the end of a function with a context containing all facts which can be derived from the semantics of the programming language and the statements in the function body. Many of these statements will be redundant but we do not anticipate this causing problems due to the length of CAS routines encountered so far and the amount of memory available to modern workstations.

At the time of writing we have specified most of the AXIOM category hierarchy, and are currently specifying entities at the domain level. We have also implemented a data-flow analyser for Aldor programs (see Section 3.1) and are currently implementing a verification condition generator based on the ideas discussed here.

References

[1] Allenby, R. *Rings, Fields and Groups.* Edward Arnold, 1991.

[2] Chalin, P. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language.* PhD thesis, Concordia University, Canada, 1995. Available as CU/DCS TR 95-12.

[3] Davenport, J., Gianni, P., Jenks, R., Miller, V., Morrison, S., Rothstein, M., Sundaresan, C., Sutor, R., and Trager, B. *Scratchpad.* Mathematical Sciences Department, IBM Thomas Watson Research Center, 1984.

[4] Dingle, A., and Fateman, R. J. Branch cuts in computer algebra. In *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation* (1994), ACM Press.

[5] Evans, D., Guttag, J., Horning, J., and Tan, Y. M. LCLint: A tool for using specifications to check code. *ACM SIGSOFT Software Engineering Notes 19* (1994), 87–97.

[6] Guaspari, D., Marceau, C., and Polak, W. Formal verification of Ada programs. In *First International Workshop on Larch* (July 1992), U. Martin and J. M. Wing, Eds., Springer-Verlag.

[7] Guttag, J. V., and Horning, J. J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[8] Jenks, R. D., and Sutor, R. S. *AXIOM: The Scientific Computation System.* Numerical Algorithms Group, Ltd. and Springer-Verlag, 1992.

[9] Jones, C. B. *Systematic Software Development using VDM.* Prentice Hall International, 1990.

[10] Jones, K. D. LM3: a Larch interface language for Modula-3. Tech. Rep. 72, SRC, Digital Equipment Corporation, June 1991.

[11] Leavens, G. T. Inheritance of interface specifications. In *Proceedings of the Workshop on Interface Definition Languages* (1994), vol. 29(8), pp. 129–138.

[12] Leavens, G. T. Larch/C++ Reference Manual. See http://www.cs.iastate.edu/~leavens/, 1997.

[13] Potter, B., Sinclair, J., and Till, D. *An introduction to formal specification and Z.* Prentice Hall International, 1991.

[14] Redfern, D. *The Maple Handbook.* Springer-Verlag, 1996.

[15] Watt, S. M., et al. A first report on the $A^\sharp$ compiler. In *ISSAC '94: Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation* (1994), ACM Press, pp. 25–31.

[16] Watt, S. M., et al. *AXIOM Library Compiler User Guide.* NAG Ltd., 1995.

[17] Wolfram, S. *Mathematica: A system for doing mathematics by computer.* Addison Wesley, 1991.