Adding Larch/Aldor Specifications to Aldor

Martin N Dunstan[†]

Department of Computer Science
University of St Andrews
North Haugh
Scotland
KY16 9SS.

Abstract

We describe a proposal to add Larch-style annotations to the Aldor programming language, based on our PhD research. The annotations are intended to be machine-checkable and may be used for a variety of purposes ranging from compiler optimisations to verification condition (VC) generation. In this report we highlight the options available and describe the changes which would need to be made to the compiler to make use of this technology.

[0]Funded by NAG Ltd

Contents

1 Introduction

It is generally accepted that specifications are a "good thing" to have in software engineering. Without a specification of the problem at hand it may not be obvious what is to be solved and in which directions we ought to proceed. Furthermore, how can one decide whether a problem has been solved correctly if we do not specified 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, for example, plain English through to mathematical notations such as Z [26] or VDM [16]. 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, to be too imprecise or even contain contradictions. 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.

In this report we describe a proposal to incorporate specifications as annotations in programs written in Aldor [29] and suggest ways in which they might be utilised. Possible uses include:

- Documentation—specifications can act as clear, concise and hopefully unambiguous documentation which are machine checkable.
- Lightweight verification—this is described in more detail later in this report and may assist users to identify mistakes in their programs which compilers are generally unable to detect.
- Incremental development—annotations of function stubs used in conjunction with a verification condition generator could be used to investigate the correctness of a program before low-level functions are actually implemented.
- Compiler optimisations—specifications could be used to select between different function implementations [28].
- Method selection—users may wish to interrogate Aldor libraries for functions which perform a particular task under specific conditions [31].

1.1 Aldor

Aldor [29] was designed as an extension language for the **axiom** [15] computer algebra system, in which computer algebra routines could be naturally and efficiently implemented. Aldor is a strongly-typed, imperative programming language that has a two-level object model of *categories* and *domains* (just like **axiom**).

1.1.1 Categories and Domains

Aldor categories are conceptually equivalent to the mathematical or logical notion of a category and are used to specify information about domains: a category defines the names and types of values (usually functions) that a domain will export to the user. Categories may be parameterised by arbitrary values, they can be joined to form new categories, and can inherit from other categories. They are closely related to Haskell type classes [14] and are a much more general form of Java or C++ virtual classes.

A domain is a type which defines zero or more exported (publicly accessible) symbols. These symbols correspond to constant values and are usually functions. 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, something which is often used when defining conditional exports. For example, domains such as `Integer` which define an ordering on their elements may claim to belong to the category `Order`; domains representing collections of values of type `T` such as `Array`, may test if `T` belongs to `Order` and provide a sorting operation if it does.

1.1.2 Types and Functions

Types and functions are first class entities which can be constructed and manipulated just like any other value in the language. Dependent types may be used in certain contexts, such as function definitions, to provide parametric polymorphism. For example, a polymorphic function to add together the elements of a list might be written as:

```
SumList(R:Ring, L:List R):R ==
{
   -- Since R is a Ring there exists an addition operator + with
   -- type (R, R)-> R and a zero value 0:R.
   local result:R := 0;

   -- Iterate over each element of L
   for elt in L repeat
      result := result + elt;

   result; -- Return value
}
```

The type of `SumList` is `(R:Ring, List R) -> R`: the first argument `R`, is a type which satisfies the category `Ring`. The type of the second argument and the return type of the function

both depend on R. The type-checking phase of the compiler ensures that this function can only be applied if the first argument is a type satisfying Ring.

1.1.3 Other Features

Other features include *post facto* extensions of existing domains, automatic garbage collection, generic iterators and inter-operability with languages such as C, C++, Fortran-77 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 inter-operability enables Aldor programs to be written to make use of existing libraries such as the NAG Fortran library or the C++ PoSSo library, and for programs written in these languages to utilise Aldor's computer algebra libraries.

1.1.4 A Simple Example

A fragment of a simple Aldor category to model domains of sets might be written:

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

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

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

This category can then be used for an implementation for a domain of sets:

```
Set(T:BasicType): SetCategory(T) == add
{
   -- Internal representation is a hash table
   Rep == HashTable(T, T);

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

      -- Return the status of the search (success/fail)
      found;
   }

   -- Other exports omitted for brevity
}
```

Watt [29] provides the definitive guide to the Aldor language while Poll and Thompson [25] describe the Aldor type system as observed at the time of writing. A strongly-typed embeddable computer algebra library called $\sum^{\text{it}}$ has been implemented in Aldor and is described in [1]; another library called $\prod^{\text{it}}$ is currently being designed to allow parallel programs to be written in Aldor [21] where the underlying architecture is abstracted away using categories.

1.2 Program Specification, Correctness and Verification

In this report we use the term "specification" for something written using a logical notation with a formal semantics; the term "annotation" is used for specifications which decorate the source code of a program to describe its behaviour and any appropriate side-conditions. One particular use of specifications is to enable a program or program fragment to be verified/proved "correct". By this we mean that the program or fragment has been formally proved to satisfy the specification.

1.2.1 Partial and Total Correctness

The notation $\{P\}\,\mathtt{C}\,\{Q\}$ states that the program fragment $\mathtt{C}$ has the pre-condition P and post-condition Q; P and Q are the specification of $\mathtt{C}$. If $\{P\}\,\mathtt{C}\,\{Q\}$ is a "partial correctness" statement then it is true, if whenever $\mathtt{C}$ is executed in a state satisfying P and if the execution of $\mathtt{C}$ terminates, then it will be in a state which satisfies Q. If $\{P\}\,\mathtt{C}\,\{Q\}$ is "totally correct" then it must be partially correct, and $\mathtt{C}$ must always terminate whenever P is satisfied.

1.2.2 Verification Conditions

The traditional approach taken to prove that $\{P\}\,\mathtt{C}\,\{Q\}$ is partially or totally correct is to reduce the statement to a set of purely logical or mathematical formulæ called "verification conditions" or "VCs" [10]. This is achieved through the use of proof rules which allow the problem to be broken into smaller fragments. For example, the rule for assignment might be:

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

Reading from the bottom, this states that to prove the partial correctness of $\{P\}\,\mathtt{v} := \mathtt{e}\,\{Q\}$ it is necessary to prove that $P \Rightarrow Q\,[\mathtt{e}/\mathtt{v}]$ where $Q\,[\mathtt{e}/\mathtt{v}]$ represents the formula Q with every occurrence of $\mathtt{v}$ replaced with $\mathtt{e}$.

As an example of VCs, the partial correctness of $\{x = 0\}\,\mathtt{x} := \mathtt{x} + \mathtt{1}\,\{x = 1\}$ relies on the verification condition $(x = 0) \Rightarrow (x + 1) = 1$. For total correctness there is the additional VC that the evaluation of $\mathtt{v} := \mathtt{e}$ terminates.

1.2.3 Lightweight Formal Methods

Our approach to verification condition (VC) generation is different to that described in the previous section. The assignment rule given was 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 the values of its arguments is very difficult and, in [7] we propose the use of lightweight formal methods to step around this problem in computer algebra systems. Rather than undertaking long and possibly intricate verification proofs, we suggest that the correctness of a procedure may be taken on trust, at least in the context of computer algebra for which a significant mathematical knowledge base exists.

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; we can then assume that Q is satisfied in the new context after `C` has terminated. Our justification for this is that we believe 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 the algorithms upon which they are based have almost certainly been well studied, sometimes even for centuries.

As a simple example, consider a program containing the annotated declaration of a function written 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);
```

Under our approach, whenever a statement such as "`a := isqrt(z)`" is encountered, we trust that the implementation of `isqrt` satisfies its specification ($\forall x \bullet x \geq 0 \Rightarrow \mathtt{isqrt}(x) = \lfloor\sqrt{x}\rfloor$) and generate the VC that $\neg(z < 0)$ must hold before the assignment. From the post-condition we can infer that $(a * a \leq z) \wedge (z < (a + 1) * (a + 1))$ is true after the assignment: a fact which may help us discharge subsequent verification conditions.

We believe that this technique is particularly useful to help detect errors in programs written using libraries of routines which have been specified with pre- and post-conditions in this way. The technique can be applied recursively: authors of library functions can use lightweight formal methods to obtain verification conditions based on even lower-level functions.

1.2.4 Using Verification Conditions

Once VCs have been generated from an annotated program, what can be done with them? Ideally one would proceed to prove that they were true but in practice this may be infeasible or even

impossible. As a facetious example one might be presented with Fermat's Last Theorem as a verification condition which is certainly difficult to prove or disprove! More realistic examples might include statements about continuity of mathematical functions or computational geometry.

Generating VCs by hand is tedious, even for tiny programs so we expect that they will be mechanically generated. Then once the VCs have been generated:

- obvious mistakes may be detected by a user quicker than a machine
- trivial VCs might be automatically discharged by the generator
- theorem provers or proof assistants might be utilised
- hand-proofs might be attempted
- the user may appeal to their specialist knowledge/authorative sources
- noted in the documentation as extra requirements
- VCs might be trusted outright

Our justification for allowing users to examine VCs without attempting to formally test their validity is that VCs might be decidable more easily by inspection than by using a mechanical checker. For example, the verification condition:

$$(\tan x) \textit{ is-continuous-on } (0, \pi)$$

is false, but it is much easier to show this by plotting the graph of $\tan x$ over the range $(0, \pi)$ than by using a general-purpose theorem prover.

Proof attempts which fail to show whether a VC is valid or invalid may indicate omissions from program annotations or from the background theory. Under certain circumstances the Cleanroom approach [5] might be applicable whereby the program is simplified to make proof attempts easier. In general the VC might simply be too difficult to validate and the additional knowledge required to to do so 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 shown to be 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.

2 Program Specification

Our work mainly centers on investigations into annotating Aldor [29] programs with Larch [13] specifications, and the generation of VCs from these annotations. In Section 2.1 we provide an introduction to the Larch approach while in Section 2.2 we introduce our work on Larch/Aldor. This is followed by Sections 2.3 and 2.4 which provide a brief overview of some related systems, specification languages and theorem provers.

2.1 The Larch Approach

Larch [13] is a two-tiered specification system primarily developed by John Guttag *et al* at MIT. In one tier users write algebraic specifications in a programming-language independent, algebraic specification language, called the Larch Shared Language (LSL). These specifications are intended to provide the background theory for the problem domain and to allow various design options to be investigated. The other tier consists of a family of behavioural interface specification languages (BISLs), each tailored to a particular programming language.

User programs are annotated using a BISL which has been designed for the chosen implementation language. The semantics of the annotations are provided by the semantics of the BISL and any LSL theories that are mentioned in the annotations. BISL specifications are primarily concerned with implementation details such as side-conditions on functions, memory allocation and pointer dereferencing: general and domain specific theory ought to be defined in LSL.

A tool called `lsl` tool is provided to perform syntax and type checking of LSL specifications. It can also convert LSL specifications into the language understood by the Larch Prover (LP). This is a proof assistant for first order logic that can, for example, be used to investigate whether LSL specifications have the properties expected of them.

2.1.1 LSL

The LSL tier allows the user to define operators and sorts (types) which can be used to model the problem domain and to provide semantics for terms appearing in the BISL annotations. Below is an example LSL specification (trait) abstracted from the Larch Handbook [13]:

```
SetBasics(E, C): trait
   includes     %%% No other traits to include
   introduces   %%% Introduce signatures of new operations
      {}: → C
      insert: E, C → C
      __ ∈ __: E, C → Bool
   asserts      %%% Axioms of these operations
      C generated by {}, insert
      C partitioned by ∈
      ∀ s:C, e, e1, e2:E
         ¬(e ∈ {});
         e1 ∈ insert(e2, s) == (e1 = e2) ∨ (e1 ∈ s);
   implies      %%% Properties which inferred from these axioms
      InsertGenerated ({} for empty)
      ∀ e, e1, e2:E, s:C
         insert(e, s) ≠ {};
         insert(e,  insert(e,  s)) == insert(e, s);
         insert(e1, insert(e2, s)) == insert(e2, insert(e1, s))
      converts
          ∈    %%% Meaning of ∈ is completely defined by the axioms
```

There are various details in this trait which won't be explained here nor will the remaining aspects of LSL specifications. Instead the user is referred to the definitive guide [13]. However, each section of the trait above has a particular meaning:

- `includes`—textually include other traits (with renaming)
- `introduces`—declares new mix-fix operators
- `asserts`—defines a set of axioms
- `implies`—statements implied by the axioms of this trait
- `converts`—operators which are completely defined by this trait

In addition, various points of interest in the trait are:

- `generated by`—all possibly values of `C` can be obtained using only `{}` and `insert`.
- `InsertGenerated({} for empty)` means that the sort `C` satisfies all the axioms of the LSL trait `InsertGenerated`, after renaming `empty` to `{}`.
- LaTeX is used for graphical symbols/operators, *e.g.* $\forall$ is written `\forall`
- the trait is parameterized by the sort names `E` and `C` representing the sorts of the elements and container respectively.

The `implies` section is used as checkable redundancy—if it is possible to show that all the statements in that section are true then we have more confidence that the axioms have been defined correctly and are sufficiently complete. Failed proof attempts may indicate the presence of mistakes or omissions in the original traits. The `implies` section can also provide extra information and lemmas which might not be obvious from axioms of the trait.

2.1.2 BISLs

Having defined (and checked) the necessary theories in LSL the user can proceed to write their program. In the ideal world users would develop the implementations in conjunction with the annotations just as one ought to do with comments. This need not always be the case however, especially when working with legacy code.

To date there are around 11 different Larch BISLS for languages ranging from CLU [30] and Modula-3 [17] to C [13] and C++ [20]. Each has been designed to investigate various aspects of imperative programming such as inheritance and concurrency as well as different development methodologies such as specification browsing [3] and interactive program verification [12].

The syntax and use of BISL specifications is essentially the same in all languages. Functions and procedures can 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.

An example Larch/Aldor program to perform integer division is given below: annotations appear as a special documentation comments in the Aldor code after the symbol `++}`. These annotations

describe the semantics of the program statement immediately following them.

```
++} ensures  result = abs(x);
local abs(x:Integer):Integer == { }

++} ensures  ((x < 0)  ⇒  (result = −1))  ∨
++}          ((x = 0)  ⇒  (result = 0))   ∨
++}          ((x > 0)  ⇒  (result = 1));
local sign(x:Integer):Integer == { }

++} requires ¬(g = 0);
++} ensures  (f = ((result.q) * g + result.r))  ∧ (abs(result.r) < abs(g));
++} modifies nothing;
integerDivide(f:Integer, g:Integer):Record(q:Integer, r:Integer) ==
{
   local quo:Integer := 0;
   local rem:Integer := f;

   ++} requires  ¬(g = 0) ∧ (quo^ = 0) ∧ (rem^ = f);
   ++} ensures   (f = (quo' * g + rem')) ∧ (abs(rem') < abs(g));
   ++} invariant f = (quo * g + rem);
   ++} measure   abs(rem);
   ++} modifies  quo, rem;
   while (abs(rem) >= abs(g)) repeat
   {
      quo := quo + sign(f)*sign(g);
      rem := rem - sign(f)*abs(g);
   }

   [quo, rem];
}

-- Test the function.
import from Integer;
ans:DivideResult == integerDivide(23, 6);
```

The **requires** clause defines the pre-condition which must hold before the statement is executed while the **ensures** clause defines the post-condition. In Larch/Aldor, these annotations are partial correctness statements (see Section 1.2.1) and cannot indicate whether or not the execution terminates. If the pre-condition is not satisfied, then nothing can be inferred about the state of the program after the statement has been executed.

Functions and operators appearing in the annotations are LSL operators *not* Aldor functions, even if they share the same name. Identifiers in the annotations represent the *logical* (LSL) values of the Aldor variables with the same name.

Since LSL values are constants it must be possible to distinguish between the different values (states) that an Aldor value may adopt. In common with other BISLs, the only states that Larch/Aldor understands are the pre- and post-states. The pre-state corresponds to the state of the program store immediately before the annotated statement is executed (denoted by appending a ^ suffix to the identifier) while the post-state corresponds to the store immediately after its execution (denoted by the ' suffix). The state of unadorned identifiers is determined from their

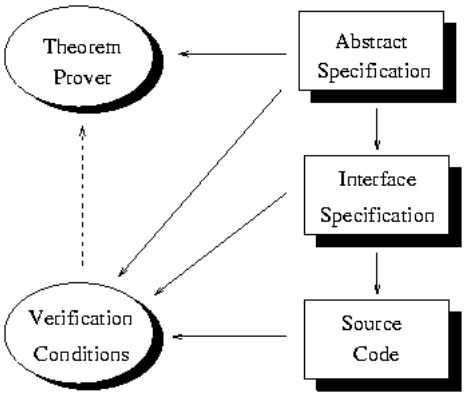
context: for example, identifiers appearing in the **requires** clause will be in their pre-state.

The special symbol `result` used to represent the value returned by a function.

2.2 Larch/Aldor

As indicated earlier, our interest lies with the application of Larch interface specifications to Aldor: our intention is to investigate the use of program annotations to improve the reliability of computer algebra routines through lightweight formal methods [7]. Aldor programs may be annotated with Larch BISL specifications similar to those in the previous example. These annotations may be used as clear and concise, machine-checkable documentation and also for verification condition generation.

The diagram below is intended to describe the development process for Larch/Aldor programs. Users begin by writing LSL specifications which provide the backgrounf theory for their problem. Next the interface specifications and Aldor source are produced, perhaps leaving some functions as stubs without a proper implementation. A separate tool is then able to generate VCs which can be analysed using the Larch Prover or some other appropriate system.



In our work so far, we have designed a Larch BISL for Aldor, and have written an LSL model of the Aldor store based on the model of Larch/C of Chalin [2]. We have also implemented a prototype lightweight VC generator written in Aldor for Larch/Aldor programs. To achieve this it was necessary to extend grammar of the Aldor compiler to enable Larch annotations to be a legal component of Aldor programs. The compiler was also modified so that it could generate an external representation of the parse tree, complete with types and Larch/Aldor annotations.

The prototype analyser uses this parse tree to generate VCs and assertions about the user's program. For example, given the annotated Aldor program in Section 2.1.2, our prototype ought to produce the VC $\neg(6 = 0)$, which is obviously true, and the assertion that:

$$(23 = ((ans.q) * 6 + ans.r)) \wedge (abs(ans.r) < abs(6))$$

Our VC generator is intended to be a proof-of-concept rather than a finished product. As a result only the top scope-level of a program is analysed and only simple data-types, such as integers, are recognised. Category and domain definitions are not analysed for VCs but annotations on exports are used by the VC generator when analysing other parts of the program. In spite of these limitations we feel that the prototype is useful as a proof-of-concept: given more time it could be extended to analyse functions and domains as well.

The prototype outputs VCs and assertions about analysed programs as LP proof scripts and so far LP has been more than capable of discharging the simple VCs presented to it. However, we believe that more interesting case studies will require the use of a more developed theorem prover such as HOL [11] or PVS [23] (see Sections 2.4.3 and 2.4.4). Changing to a different theorem proving system ought to be relatively easy due to the modular nature of the prototype.

Closely related to ours is the work of Kelsey [7] who is specifying a significant part of the **axiom** category hierarchy in LSL. This is essential if more complex computer algebra examples are investigated with the VC generator since we need the underlying LSL theories to provide meanings for the operators used in Larch/Aldor interface specifications.

2.3 Related Work

2.3.1 Extended Static Checking

The Extended Static Checking (ESC) system [6] was developed at DEC SRC to provide automatic machine checking of Modula-3 programs. The techniques developed for this project are now being applied to Java. The goals of ESC include the following:

- detecting violations of array bounds and `NIL` pointer dereferencing
- detecting deadlocks and race conditions in concurrent programs
- simple annotations yet powerful, *e.g.* $\forall i : 0 \leq i < n : a[i] \neq \texttt{NIL}$
- fast (comparable to the compiler), non-interactive checking
- development of a theorem prover to achieve the above

2.3.2 ProofPower

This is a commercial tool developed by the High Assurance Team (HAT) at ICL [19]. It is based on the HOL [11] theorem prover (see Section 2.4.4) and the Z notation [26] (see Section 2.4.1) for a subset of the Ada programming language.

- functional requirements of programs are often presented to HAT as documents containing Z specifications. These can be checked and 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. These can be discharged via formal or informal arguments as required (*e.g.* with ProofPower).

An example program extracted from [19] is given below. Everything before $\sqsubseteq$ is the original Compliance Notation while the remainder is the Ada implementation:

```
procedure EG(X:in out INTEGER; Y: in out INTEGER)
is
        A:INTEGER;
begin
        ΔX, Y [X = X0 * X0 + Y0 * Y0 ∧ Y = 2 * X0 * Y0]
⊑
        A := X + Y;
        Y := X * Y;
        Y := Y + Y;
        X := A*A - Y;
```

from which the following trivial VC is generated:

$$
\begin{array}{ll}
\vdash & \forall X : INTEGER;\ Y : INTEGER \bullet \\
 & (X + Y) * (X + Y) - (X * Y + X * Y) = X * X + Y * Y \\
\wedge & X * Y + X * Y = 2 * X * Y
\end{array}
$$

2.3.3 Eiffel

In the Eiffel [22] programming language, specifications containing pre- and post-conditions are an integral part of the language syntax. The compiler can 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.

In the example Eiffel program below (from [22]), the `require` clause defines the pre-condition of the function while the `invariant` clause specifies the loop invariant of the Eiffel `from`

loop. The `variant` clause defines a value that must decrease each time the loop body is executed and must be non-zero when the loop terminates. This is equivalent to the `measure` clause in Section 2.1.2. Post-conditions are be introduced with an `ensure` clause while informal specifications appear as comments.

```
gcd(a, b:INTEGER):INTEGER is
   require a > 0; b > 0
   local   x, y:INTEGER
   do
      from      x := a; y := b
      invariant x > 0; y > 0; -- ∧ gcd(x, y) is equal to gcd(a, b)
      variant   max(x, y)
      until     x = y
      loop
         if x > y then x := x - y; else y := y - x; end
      end
      Result := x
   end
end;
```

Note that 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 check the assertions at runtime and generate exceptions when they are violated is certainly better than the facilities provided by most other programming languages.

2.3.4 Extended ML

Extended ML [27] 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 (reification [16]) 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.

Below is an example of a stack signature taken from [27]:

```
signature STACK =
   sig
      structure Obj : OBJ
      type stack

      val empty : stack
      val push  : Obj.object * stack → stack
      val pop   : stack              → stack
      val top   : stack              → Obj.object

      axiom pop(push(a, s)) = s
      axiom top(push(a, s)) = a
   end
```

The `axiom` statements define the algebraic specification of the behaviour of the `top`, `pop` and `push` functions while the remainder of the signature is Standard ML.

Note that the specification is incomplete since it does not specify the behaviour of `top` or `pop` when applied to an empty stack. Also note that the functions can be used as logical predicates since ML is a functional programming language. This is not possible in an imperative language such as Aldor, hence the need for a BISL such as Larch/Aldor.

2.3.5 Speckle

This is a compiler which was designed to investigate how specifications could be used make programs run faster. Speckle [28] accepts CLU programs containing different implementations of procedures which are annotated with Larch-style interface specifications. An additional language construct `special when` allows the programmer to specify the conditions under which specialised implementations may be used in place of the default implementation.

Speckle uses the interface specifications to determine the program state at each procedure application. If a procedure has a specialised implementation then the compiler attempts to discharge each of the conditions associated with the specialisation. If a condition is satisfied then the default implementation is replaced with the specialised one. This enables the user to provide domain specific optimisations without the need for (possibly expensive) runtime tests.

2.3.6 LcLint

Many of the Larch BISLs do not have any program analysis tools associated with them—they are primarily used as clear and concise documentation. The exceptions are Larch/Ada (see Section 2.3.7), Larch/Aldor (see Section 2.2) and Larch/C which is described here.

LcLint [9] is a static checker for C programs written using Larch/C and other more specialised annotations. The only BISL specification checks that are made by LcLint are those relating to violations of the **modifies** clause. These violations correspond to attempts by the implementation of the function to mutate the values of an object which is not listed in the **modifies** clause.

In addition LcLint has been designed to understand special comments describing the behaviour of the user's program. Examples include the marking of function parameters as `out` when they are used to return values to the caller and statements about whether an abstract data type (ADT) is mutable or not. Since C does not provide support for abstract data types, LcLint encourages an ADT style of programming and is able to detect violations of it. For example, an implementation of a list data-type might be declared as *abstract* and provide a set of interface procedures. LcLint will then warn the user if attempts are made to access the list representation directly.

2.3.7 Penelope

Penelope [12] is a syntax-directed editor designed for the interactive development and verification of Larch/Ada programs. The user enters their program and annotations in the editor which automatically generates verification conditions. At any point the user can attempt to prove verification conditions or modify the program or specification, perhaps when a mistake is detected. The editor automatically adjusts the set of verifications conditions following each change.

An example from Guaspari [12] is:

```
--| WITH TRAIT SortingAndSearching
PACKAGE searching
   TYPE intarray IS ARRAY(integer) OF integer;

   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;

   -- ... Rest of package omitted ...
END searching;
```

The `WITH` statement indicates that the LSL trait `SortingAndSearching` provides semantics for the annotatons which follow the Ada comment marker `--|`. The `IN` clause defines the pre-condition of the function while the `RETURN` clause defines the post-condition.

2.4 Program Specification and Theorem Provers

Closely related to the above systems are the use of specification languages and theorem provers. Here we briefly describe two well-known specification languages, Z [26] and VDM-SL [16], and two well-known theorem provers, PVS [23] and HOL [11].

2.4.1 Z Notation

Z [26] is a formal specification language or notation based on Z-F set theory and first-order predicate logic. In addition, Z specifications may contain state variables and enabling imperative programs to be specified relatively easily. Unfortunately Z makes heavy use of special symbols which can make specifications difficult to understand. However, it is common for Z specifications to be embedded in explanatory prose thereby enhancing the reader's comprehension.

A simple Z specification of the `sign` function from Section 2.1.2 might be written as in the

example below. The specification or schema is given the name *SignOf* and is split into two sections. The first section declares the name and type of the symbols being specified while the second section provides the axioms associated with these symbols:

$$
\begin{array}{|l}
\hline
SignOf \\
\hline
sign : \ \mathbf{Z} \to \mathbf{Z} \\
\hline
\forall\, n : \mathbf{Z} \bullet \\
\quad (n > 0 \Rightarrow sign\ n = 1) \land \\
\quad (n = 0 \Rightarrow sign\ n = 0) \land \\
\quad (n < 0 \Rightarrow sign\ n = -1) \\
\hline
\end{array}
$$

Various tools exist to assist with the type checking and development of Z specifications. Several are based around a LaTeX-style input language (*e.g.* FUZZ) while others such as Cogito use a much simpler format which may be less intimidating to the novice user. See also Section 2.3.2

2.4.2 VDM

Strictly speaking, VDM [16] is the name of a development method—the specification language is called VDM-SL and allows the user to specify mathematical models using a functional-style language as well as via pre- and post-conditions; new types may be introduced based on built-in types such as records and sequences. In contrast to other systems, VDM-SL is based on a three-valued logic which allows users to work in the presence of undefined values which can appear in real-world programs. However, this logic can make proof-attempts more difficult.

Below is an example of a VDM-SL specification taken from a case study by the author on quantum mechanics showing both specification styles:

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

$$
\begin{array}{l}
\mathrm{Nnl}\ (n : \mathbf{Z},\ l : \mathbf{Z})\ z : \mathbf{R} \\
\quad \mathbf{pre} \quad (n - l \geq 1) \land (n + l \geq 0) \land (n \neq 0) \\
\quad \mathbf{post} \quad z = N_{nl}
\end{array}
$$

2.4.3 PVS

PVS [23] is verification system based on classical, typed higher-order logic with support tools and a theorem prover developed at the Stanford Research Institute (SRI). The available types include built-in types such as booleans, integers and reals as well as type constructors for functions, sets, records *etc*. Other features which are particularly useful in the context of Aldor are

dependent types, parameterised theories and predicate sub-typing. An example PVS specification taken from a PVS tutorial is:

```
function_properties[D, R:TYPE]: THEORY
BEGIN
   f, g      : VAR [D → R]
   x, x1, x2 : VAR D
   y         : VAR R

   injective?(f):bool  = (FORALL x1, x2: (f(x1) = f(x2) ⇒ (x1 = x2)))
   surjective?(f):bool = (FORALL y: (EXISTS x: f(x) = y))
END function_properties

finite[t:TYPE]: THEORY
BEGIN
   IMPORTING function_properties

   is_finite_type:bool = (EXISTS (n:nat), (f:[upto[n] → t]): surjective?(f))

   is_finite_type_alt: LEMMA
      is_finite_type IFF (EXISTS (n:nat), (g:[t → upto[n]]): injective?(g))
END finite
```

2.4.4 HOL

HOL [11] is an interactive theorem proving environment for higher-order logic and is highly programmable through the functional programming language ML [24].

An example of a recursive definition of `plus:num -> num -> num`, a curried addition function taken from an introduction to HOL is given below. The hash marks (#) are the HOL prompt:

```
# let PLUS = new_prim_rec_definition
#            (‘PLUS‘,
#               "(plus 0 n = n) ∧
#                (plus (SUC m) n = SUC(plus m n))");;
PLUS = |- (!n.  plus 0 n = n) ∧
           (!m n.  plus(SUC m)n = SUC(plus m n))
```

where `!n. E` represents the universally quantified statement $\forall n\, E$ and where `SUC` is the successor operator.

3 The Proposal

We propose to add Larch-style annotations to Aldor programs following the work described in [7] and [8]. We would like users to be able to annotate Aldor programs with specifications mainly describing the pre- and post-conditions of program fragments. In addition we may wish

to follow the Larch [13] and VDM [16] approach of highlighting any potential modifications to client-visible state; we might also wish to decorate loops with invariants and termination requirements. These annotations may be used in various ways as described below.

3.1 Program Annotations

As described in the introduction of this report, we can use the annotations as clear and concise documentation. However, this does not require any special modifications to the Aldor compiler or the way in which users write their programs. More importantly it doesn't provide any tangible benefits to the user in the form of warnings or error messages.

We would like to take the concept further and allow the compiler and other tools to utilise the annotations. This might be just syntax and type-checking but could go further:

- extended lint-style checks (*c.f.* LcLint [9])
- verification condition generation (*c.f.* Sections 1.2.2 and 1.2.3)
- compiler optimisations via specifications (*c.f.* Speckle [28])
- runtime assertions (*c.f.* Eiffel [22])
- automatic test-case generation

We believe that compiler or tool support of this is essential—programmers often adopt the path of least resistance: unless annotations will have a practical benefit, they are less likely to be used. The tool support must be easy to use and relatively fast otherwise programmers will avoid the annotations and lose any associated benefits. Ideally the checking and analysing annotations, must be comparable in execution time to the compilation itself.

3.2 Design Decisions

There are a number of options to be considered for this project—these are outlined below and then discussed in more detail in the sections that follow:

- Should we annotate existing Aldor library routines?
- Do we use auxiliary specifications (two-tiers)?
  - should we provide sample specifications?
  - should we incorporate them into Aldor?
- Are verification conditions useful?
  - which language features do we support?
- Is a theorem prover necessary, and if so, which one?
  - interactive/non-interactive?
  - tidying-up VCs or for full-scale proof attempts
  - embedded or loosely coupled

- Are specification-based optimisations safe?
- Should we provide support for external tools?

3.2.1 Annotating Library Routines

Since we are providing users the ability to add annotations to their programs, we ought to consider annotating existing library routines. To some extent the choice of routines to annotate depends on whether auxiliary specifications will be supported. If so, our annotations will need to be based on such specifications which may place restrictions on what can be done—we must decide which routines would be interesting, useful or tractable to specify and annotate.

3.2.2 Auxiliary Specifications

Auxiliary specifications can significantly extend the scope of program annotations and places them on a firm, logical basis. In the Larch approach (see Section 2.1) LSL specifications provide definitions and semantics for operations and sorts which can then be utilised in the interface specifications. Similarly, a user of the ProofPower system (see Section 2.3.2) is able to refer to specifications written in Z and Compliance Notation.

The ESC system (see Section 2.3.1) demonstrates that auxiliary specifications are not essential but the drawback is that users of the system are restricted to the domains selected by the system designer such as array bounds checking or `NIL` pointer dereferencing. There appears to be little scope for future extensions except those provided by the ESC developers.

Supporting auxiliary specifications means that we may need to provide a library of useful specifications such as the Larch Handbook [13] or those of the **axiom** category hierarchy [18].

Given the two-tier nature of the Aldor object model, we could consider making LSL-style algebraic specifications part of the language. This might be in the form of a separate primitive directly corresponding to an LSL trait or in the form of Extended-ML style axioms. By incorporating algebraic specifications in the language, we could make use of the Aldor inheritance and parameterisation mechanisms which LSL currently lacks.

We also have the issue of providing theorem proving support to help users to debug and check their auxiliary specifications.

3.2.3 Verification Conditions

Is verification condition generation is a useful direction to take? If so, what do we expect the user to do with verification conditions—if they wish to attempt to show that they are satisfied

then some kind of theorem proving support will be necessary.

We also need to decide which features of Aldor we ought to concentrate on: the implementation of a verification condition generator is time consuming and Aldor has a wide range of sophisticated features that are not found in languages such as C.

3.2.4 Choice of Theorem Prover

Some of the options discussed here may benefit from, or even require, the use of theorem proving technology. For debugging auxiliary specifications (Section 3.2.2), an interactive theorem prover or proof assistant with a short learning curve is desirable; a limited form of programmability might be beneficial. Alternatively if we intend to use the theorem prover to simplify verification conditions or to attempt discharge some of them automatically, then an interactive system may be a disadvantage.

Other issues include: ease-of-use for interactive theorem provers, whether or not they are still being developed, and the size of the user-base or support groups. If the system is to be used as a "black-box" then we need to decide how tightly coupled to the Aldor compiler it will be: embedded as library routines or running as a separate process? In addition we need to bear in mind our relationship with the developers of the theorem prover and consider the implications of distributing such a system with Aldor. It is worth noting that a theorem proving system that is no longer being developed might actually be a benefit: its developers may be more willing to allow their system to be distributed in conjunction with another system. We might consider writing our own theorem prover but this is a full-scale project in itself.

3.2.5 Specification-based Optimisations

The optimisations of the Aldor compiler, as with other compilers, are intended to be safe. It is not acceptable for the optimiser to make code transformations that alter the behaviour of a program in ways that the programmer or user does not expect.

By allowing the compiler to make optimisations based on specifications we introduce the possibility that faulty specifications may lead to invalid optimisation decisions; such mistakes could be extremely difficult to track down and eliminate. This type of system would need to keep the user informed about which implementations were selected and why.

3.2.6 External Tool Support

By incorporating any of the above options as extra modules or phases in the compiler, we make life easier for the user but may restrict what they can achieve. If we modify the compiler so that

it produces an annotated abstract syntax tree or control-flow graph in addition to an executable, then users can develop tools to achieve satisfy their own specific needs.

4 Compiler Modifications

In this section we describe the modifications that would need to be made to the existing Aldor compiler to support machine-checkable annotations. We only consider the work specifically involved with the annotations and ignore issues such as verification condition generation or specification-based compiler optimisations.

If the changes described below are implemented, then the compiler will be able to perform syntax and type checking of annotated Aldor programs. In addition, annotations of exported symbols such as functions will be stored in libraries along with the type information. This will enable annotations to be used even when the original source code is unavailable.

The time-scales associated with each of these changes are based on the assumption that they will be implemented by someone already familiar with the current Aldor compiler. Hopefully these estimates are not too dissimilar to those needed to implement these changes in compilers of other programming languages.

4.1 Syntax Analysis (*2–3 weeks*)

We assume that the syntax of the annotations will be simple compared to Aldor. Annotations will consist of a sequence of clauses, each beginning with a label such as **requires** or **ensures**. Depending on the clause type, the body will either be a single logical expression or a list of symbols/identifiers. Logical expressions may contain function applications, identifiers, program literals and the usual logical connectives.

A suitable lexical analyser for such annotations could be implemented very easily in most languages: we have already written one using approximately 60 lines of Aldor. The syntax analyser could be constructed using a tool such as YACC, or by hand using the recursive descent parsing techniques of Davie [4].

4.2 Type Checking

Using auxiliary specifications to define new operators and their semantics makes type inference more difficult due to the increased overloading of symbols. For example, in the annotation of the program fragment below:

```
++} modifies nothing;
++} ensures result = (a + b);
plus(a:Integer, b:Integer):Integer ==
   (a + b);
```

the identifiers `a` and `b` represent the values in the underlying logic, of the formal parameters of the Aldor `plus()` function. Likewise the `+` operator in the specification is an operator from the underlying logic that operates on suitable values from this logic. This means that we have two `+` operators with the same signature:

`+`: (`Integer`, `Integer`) → `Integer`	*from* `Integer`
`+`: (`Integer'`, `Integer'`) → `Integer'`	*from the underlying logic*

where `Integer'` represents the integer type from the underlying logic used to model the Aldor domain `Integer`.

Two possible solutions to this problem are considered in the following sections. The first is to make use of syntactic sugar and implicit coercions, while the second is to mangle identifiers.

4.2.1 Implicit Coercions (*1–2 weeks*)

With this method we treat identifiers in annotations as syntactic sugar for a coercion between an Aldor domain and a sort in the underlying logic. Thus the value of the **ensures** clause in the previous example might be interpreted as '`result = preState(a) + preState(b)`'.

By adding domains to the Aldor libraries corresponding to LSL sorts, this could be type-checked in the same was as any other Aldor expression. For example, given the domain:

```
+++ 'LSLinteger' is the special Aldor domain corresponding
+++ to the LSL sort 'Integer'.
LSLinteger:  with {
   preState:  Integer -> %;
   preState:  Integer -> %;
   +:  (%, %) -> %;
   -- Other exports omitted for brevity.
} == add { ... }
```

the type-checker can infer that '`result`' is of type '`LSLinteger`' if '`a`' and '`b`' are of type '`Integer`'. Since these domains are only used for type checking the annotations, the implementation of their exports is irrelevant.

Ideally these domains would be constructed internally by the compiler from LSL specifications. Thus they would only be seen by the type checker and not by the parser or code generators.

4.2.2 Name Mangling (*2–3 weeks*)

An alternative solution to the problem of type-inferring annotations, is to mangle the names of operators and sorts in specifications so that they do not clash with names elsewhere in the source code. This would allow the operators to appear in the Aldor symbol table and be used without the type-inference engine realising that they are from a different language.

In principle this is straightforward but we must ensure that the mangled names do not clash with identifiers in the user's program and do not contain characters that would cause problems for the compiler. Also the user must never see these names in their mangled form.

4.2.3 LSL Theories (*2–3 weeks*)

Whichever approach is adopted, the Aldor symbol table must be populated with definitions taken from the auxiliary specifications. These are would be introduced via the **uses** clause in an annotation and would be equivalent of the Aldor `import` statement.

4.3 Code Generation (*2–5 weeks*)

During the code generation most nodes can be stripped of their annotations leaving the existing functions to continue as normal. This is possible because once the program has been analysed, the only annotations of interest are those on attached to exported symbols.

Exported symbol definitions containing specifications are harder to deal with since we will want to make their annotations available to users of the libraries. However, it ought to be possible to use the existing mechanism of recording documentation comments in libraries so that annotations are stored as well.

5 Summary

This report describes a proposal for extending the Aldor [29] programming language with annotations in the style of Larch [13] BISLs based on our PhD work concerning lightweight formal methods for computer algebra systems [8] and our prototype verification condition generator.

In Section 1.1 we introduced the reader to Aldor with a brief description of its salient features such as categories and domains. Then in Section 1.2 we introduced the topics of program specification, correctness and program verification, including in Section 1.2.3 the subject of lightweight formal methods and the uses that verification conditions can be put to.

Then in Section 2.1 we provided a brief introduction to the Larch approach to program specification and formal methods, followed by a description of Larch/Aldor: a Larch BISL developed by the author. Sectionsection:specification:related describes related work while Section 2.4 looks at two well known specification languages and two well known theorem provers.

The proposal for extending Aldor and its compiler is laid down in Section 3: we look at the design decisions involved, questions regarding the choice of theorem prover and issues of external tool support. Then in Section 4 we examine three important phases of a compiler that must be modified to support our proposed extensions. Approximate time-scales for completing this work are given under the assumption that it would be undertaken, full-time, by a compiler developer.

In light of this report and our work described in [8], we believe that it would be relatively easy to extend the Aldor compiler to support Larch-style annotations with between six and eleven weeks work to produce a robust and reliable system. However, this does not take into account the time required to develop the necessary background theories and specifications for the base library. Nor does it include the more time consuming task of a verification condition generator: a prototype verification condition generator written in Aldor took about six months to implement [8].

I would like to thank NAG for their funding during this work and for providing access to the compiler source. Without this our work would have been much more difficult.

A Program Verification

In this section we provide some additional background information relating to the implementation of verification condition generators and related topics following from the discussions of Section 3.2.

A.1 Verification Condition Generation

The basic implementation of a verification condition generator is relatively straightforward once some basic design decisions have been made:

- forwards or backwards analysis
- internal representation of programs
- lightweight or full verification
- which language to expression verification conditions in

The difficult parts of an implementation may be summarised as:

- removing implicit time-dependence
- the internal representation of the program
- storage and manipulation of the program state
- defining the semantics of language features

To understand the meaning of an imperative program it is essential to know the order in which statements will be executed and the behaviour of assignments. This complicates any reasoning about imperative programs since most logics are naturally based around the functional programming paradigm. Thus whereas the value of an identifier in an imperative program depends on which statements have been executed previously, the value of a logical identifier is the same at all times.

A.1.1 Backwards Analysis

This is the approach which is commonly adopted—using semantics of the programming language, the post-condition is "pushed" backwards through the code until it reaches the top. The verification condition is that the resulting weakest pre-condition is satisfied by the specified pre-condition. The advantage of this method is that the formulae generated are compact since at each stage we only need to know the conditions associated with the current statement which are sufficient to prove the post-condition. The disadvantage is that simple programming language constructs such as the assignment operator can cause subtle difficulties. A simple example of this process is given below:

$$
\begin{array}{c|ccc}
\textbf{pre} & \{true\} & \{true\} & \{true\} \\
 & & & \{42 > 23\} \\
 & \texttt{x := 42;} & \texttt{x := 42;} & \texttt{x := 42;} \\
 & & \{x > 23\} & \{x > 23\} \\
 & \texttt{y := 23;} & \texttt{y := 23;} & \texttt{y := 23;} \\
\textbf{post} & \{x > y\} & \{x > y\} & \{x > y\}
\end{array}
$$

which yields the simple verification condition $true \Rightarrow (42 > 23)$.

A.1.2 Forwards Analysis

In contrast, forwards analysis pushes the pre-condition through the program statements using the post-conditions to determine the program state at each stage. When the process reaches the end of the program the resulting formulae represents the strongest post-condition and the verification condition is that this satisfies the original post-condition. The disadvantage of this method is that we need to store everything we know about each stage of the program, a lot of which might not be relevant to the final post-condition. We also need to distinguish between the different states that a variable may be in following each assignment or mutation as described earlier.

The previous example reworked for the forwards approach is as follows:

$$
\begin{array}{c|ccc}
\textbf{pre} & \{true\} & \{true\} & \{true\} \\
 & \texttt{x := 42;} & \texttt{x := 42;} & \texttt{x := 42;} \\
 & & \{x = 42\} & \{x = 42\} \\
 & \texttt{y := 23;} & \texttt{y := 23;} & \texttt{y := 23;} \\
 & & & \{x = 42 \wedge y = 23\} \\
\textbf{post} & \{x > y\} & \{x > y\} & \{x > y\}
\end{array}
$$

which yields the simple verification condition $(x = 42) \wedge (y = 23) \Rightarrow (x > y)$. Note that a longer program would have generated a verification condition with many more terms whereas the backwards approach would not.

A.1.3 Lightweight Approach

The solution to the problem of time-dependence that we have adopted in a prototype lightweight verification condition generator, is to add an explicit state reference to all identifiers. This is not quite as simple as it first seems however. For example, the program from Section 2.3.2 might be represented internally with the explicit state annotations appearing as comments as follows:

```
++} requires true
++} ensures x' = x^ * x^ + y^ * y^ ∧ y' = 2 * x^ * y^
EG():() ==
{
   free x, y:Integer;    -- declare x0, x1, y0, y1, y2 :  → Integer
   local a:Integer;      -- declare a0, a1 :  → Integer

   a := x + y;           -- assert (a1 = x0 + y0)
   y := x * y;           -- assert (y1 = x0 * y0)
   y := y + y;           -- assert (y2 = y1 * y1)
   x := a*a - y;         -- assert (x1 = a1 * a1 − y2)
}
```

Any invocation of `EG()` will generate the trivial verification based on the pre-condition of this function that *true* is true. We can then assume that the post-condition describes the program state after `EG()` has terminated. For example:

```
local x, y:Integer; -- declare x0, x1, y0, y1 :  → Integer
x := 3;             -- assert x0 = 3
y := 1;             -- assert y0 = 1
EG();               -- assert x1 = x0 * x0 + y0 * y0 ∧ y1 = 2 * x0 * y0
```

At the end of this program fragment, the context contains the following information:

- $x_0 = 3$
- $y_0 = 1$
- $x_1 = x_0 * x_0 + y_0 * y_0 \wedge y_1 = 2 * x_0 * y_0$

If desired, simplification of the context yields:

- $x_0 = 3$
- $y_0 = 1$
- $x_1 = 11$
- $y_1 = 6$

As noted in other sections, this context will grow as each program statement is analysed and may enable us to quickly discharge some other verification conditions in the future.

References

[1] 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*.

[2] 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.

[3] 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.

[4] DAVIE, A. J. T., AND MORRISON, R. *Recursive descent compiling*. Ellis Horwood, 1981.

[5] DECK, M. Cleanroom software engineering and "old code"—overcoming process improvement barriers. In *Proceedings of the 1995 Pacific Northwest Software Quality Conference* (1995).

[6] DETLEFS, D. L. An overview of the extended static checking system. Nov. 1995.

[7] DUNSTAN, M., KELSEY, T., LINTON, S., AND MARTIN, U. Lightweight formal methods for computer algebra systems. In *ISSAC* (1998).

[8] DUNSTAN, M. N. *Larch/Aldor—A Larch BISL for AXIOM and Aldor*. PhD thesis, Mathematical and Computational Sciences, University of St Andrews, Sept. 1999.

[9] 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.

[10] GORDON, M. J. C. *Programming language theory and its implementation*. Series in Computer Science. Prentice Hall International, 1988.

[11] 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.

[12] 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.

[13] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification*, first ed. Texts and Monograps in Computer Science. Springer-Verlag, 1993.

[14] HUDAK, P., JONES, S. L. P., WADLER, P., ET AL. A report on the functional language Haskell. *SIGPLAN Notices* (1992).

[15] JENKS, R. D., AND SUTOR, R. S. *AXIOM: the scientific computation system*. NAG Ltd., 1992.

[16] JONES, C. B. *Systematic Software Development using VDM*, second ed. Computer Science. Prentice Hall International, 1990.

[17] 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.

[18] KELSEY, T. *Formal Methods and Computer Algebra: A Larch Specification of AXIOM Categories and Functors*. PhD thesis, Mathematical and Computational Sciences, University of St Andrews, Dec. 1999.

[19] KING, D. J., AND ARTHAN, R. D. Development of practical verification tools. *The ICL Systems Journal 1* (May 1996).

[20] 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.

[21] MANNHART, N. $\prod^{\text{it}}$: an aldor library to express parallel programs. Available on the WWW at `www.inf.ethz.ch/personal/mannhart`.

[22] MEYER, B. *Object-Oriented Software Construction*. Computer Science. Prentice Hall International, 1988.

[23] 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.

[24] PAULSON, L. C. *ML for the working programmer*. Cambridge University Press, 1993.

[25] 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.

[26] POTTER, B., SINCLAIR, J., AND TILL, D. *An introduction to formal specification and Z*. Prentice Hall International, 1991.

[27] 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.

[28] VANDEVOORDE, M. T., AND GUTTAG, J. V. Using specialized procedures and specification-based analysis to reduce the runtime costs of modularity. In *Proceedings of the 1994 ACM/SIGSOFT Foundations of Software Engineering Conference* (1994).

[29] 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.

[30] WING, J. M. A two-tiered approach to specifying programs. Tech. Rep. LCS/TR–299, Laboratory for Computer Science, MIT, May 1983.

[31] 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.