72

LM3: a Larch Interface Language for Modula-3 A Definition and Introduction

Kevin D. Jones

June 13, 1991

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

LM3: a Larch interface language for Modula-3
A definition and introduction
Version 1.0

Kevin D. Jones

June 13, 1991

Abstract

This report describes a Larch interface language (LM3) for the Modula-3 programming language. LM3 is a complete example of a Larch interface language and addresses areas previously ignored in interface language definition, such as the specification of non-atomic procedures and object types.

We give a complete definition of the syntax and illustrate it with some straightforward examples. We also give translation functions from LM3 specifications to Larch Shared Language traits and show their use for type checking. Finally, we present example specifications of standard Modula-3 interfaces.

To remove the possibility of misunderstanding, this report presents LM3 using its base syntax and does not use any syntactic sugar. In practice, such sugar is convenient and the checker accepts a sugared form as well as the raw form presented here.

Contents

Chapter 1

Introduction

1.1 Background

Larch provides a family of specification languages that may be used to specify program interfaces, plus a collection of tools to aid in constructing correct specifications. Larch specifications are given in two parts:

1. the *Shared Language tier*, which uses an algebraic specification language to describe the properties of the basic data types and operators used in the specification. The Larch Shared Language (LSL)[6] is common to all Larch specifications.

2. the *interface language tier*, which is specifically related to the programming language being used. An interface language contains constructs that are appropriate to the programming language and uses the traits specified in the LSL tier to describe the properties of the interface.

This report describes an interface language (LM3) that is designed for use with the Modula-3 language[4]. It is assumed that the reader is already familiar with Modula-3, LSL, and the general ideas of interface specification (as in, say, [13]).

Previous publications have documented Larch interface languages, for example [12]. LM3 follows the general style of this previous work, but addresses several additional features that are becoming common in new programming languages.

- Modula-3 allows higher order procedures (i.e., those which take procedure parameters or return procedure results). Since procedures are represented by their specifications rather than by their values, we need a way of associating specifications with such parameters.

- Modula-3 allows object type hierarchies. This means LM3 has to inherit specifications from supertypes.

- Modula-3 supports concurrently executing threads of control, therefore, LM3 has to be able to specify non-atomic procedures.

LM3 addresses each of these areas. Where possible, the extra features follow the style and philosophy of previous Larch work.

1.2 The relationship between Modula-3 and LM3

A Modula-3 module that provides an externally usable interface usually consists of two files:

1. an *interface* file, with the extension .i3, which defines the exported definitions of the module;

2. an *implementation* file, with the extension .m3, which gives the full code of the module.

LM3 specifications are placed in the *interface* file. Clients of the module see only the .i3 file. All of the information necessary for understanding and using the module should be presented in this file. Without LM3, the interface is usually supplemented by comments describing the intended action of the procedures, and so forth. From the point of view of the clients, the LM3 specification provides a more precise description of the functionality of the interface and can replace some of the detailed textual comments, although general comments should still be used to supplement the formal text. From the point of view of the programmers of the .m3 file, the LM3 specification provides a contract to which they must implement.

LM3 annotations decorate standard Modula-3 as *pragmas*. The main intention of *pragmas*, according to the Modula-3 report, is to provide 'hints to the implementation; they do not affect the language semantics.' Since unrecognized *pragmas* are ignored by the compiler, they provide a convenient way of attaching the specification information.

We believe that specifications should be kept with the programs they are intended to describe. LM3 interface specifications are legal Modula-3 interfaces. By making the specification an integral part of the program, we intend to remind the programmer of its existence at all stages in the life of the interface. Previous Larch interface languages have placed the specifications within comments but doing this has tended to de-emphasize their importance in the minds of some. We expect pragmas to be taken more seriously.

Chapter 2 introduces the constructs of LM3 and illustrates their use with simple examples. The complete syntax of LM3 is given in Chapter 3. Chapter 4 defines the LSL traits used by LM3 and gives the translation functions for LM3 constructs. Chapter 5 gives examples of complete LM3 interface specifications.

Chapter 2

The LM3 specification language

An LM3 interface specification consists of a Modula-3 interface definition together with some annotations within *pragma* brackets. A specification contains type specifications, variable and constant declarations, procedure specifications and an interface invariant. In the following sections, we consider these in turn.

We introduce the following grammatical conventions:

- terminal symbols — **term**
- LM3 and Modula-3 keywords — `KEYWORD`
- non-terminals — *nonTerm*
- foo,* means zero or more foo's separated by a ,
- foo;$^+$ means one or more foo's separated by a ;
- [foo] means zero or one foo
- extensions to the Modula-3 grammar are indicated as <u>foo</u>

2.1 Interfaces

The top-level unit in an LM3 specification, like that of a Modula-3 program, is the interface. A Modula-3 interface is annotated with a list of traits that define the meaning of the symbols used within predicates to specify the functionality of the interface components. Any renaming necessary to remove ambiguity must be done here, using LSL mechanisms.

An interface specification has the form:

```
interface       ::=  INTERFACE ident ; [traitUse] imports*
                     [intConstraints] declaration* END ident .
traitUse        ::=  <* USING traitRef,+ *>
imports         ::=  [FROM ident] IMPORT ident,+ ;
intConstraints  ::=  <* initial *> | <* invar *> | <* initial invar *>
declaration     ::=  constDecl | varDecl | typeDecl | exceptionDecl
                     | procDecl | privateVarDecl | ...
initial         ::=  INITIALLY lm3Predicate
invar           ::=  INVARIANT lm3Predicate
```

There may be an initial condition and an invariant associated with the interface. In an interface that declares variables, the initial condition constrains the initial value of the variables. The invariant would normally be used to state relationships that must always be maintained between these variables. It may also be used to specify relationships between procedures within the interface.

Since grammar fragments are not the most enlightening way of understanding a language, we develop a simple example[1] as we introduce each new construct. A skeleton of the interface is given as:

```
INTERFACE Stack;
<* USING Stack(Real for E, RealStack for C) *>

(* declarations of exported type and procedures - see below *)
END Stack.
```

The `USING` clause associates all symbols used in the specification with those found in the trait named Stack, with the appropriate renaming. All interfaces implicitly use a trait *LM3Trait* which gives the definitions of the primitive operations of Modula-3. See Chapter 4. The trait Stack, from the Larch Shared Language Handbook[8], is given in Appendix A.

[1] Our excuse for using the ubiquitous stack example is that we are following the Modula-3 Report, which uses Stack as its example of an interface.

2.2 Declarations

Declarations in Modula-3 interfaces include constants, types, variables, exceptions and procedures. In this section, we describe the LM3 specifications for each since the form of the specification depends on the kind of declaration.

2.2.1 Constants

An interface may export any number of constants. The grammar for constant declarations is:

$$constDecl \quad ::= \quad ident\ [\mathbf{:}\ type] = constExpr$$

A constant declaration for LM3 is just a Modula-3 constant declaration, with the restriction that the *constExpr* must be a term of the LM3 expression language.

For example, if we wished to give an upper bound to the stack, we could write:

```
CONST MaxSize = 100;
```

2.2.2 Variables

LM3 adds no extra information to the declaration of exported variables. Any restrictions may be placed in either the type or the interface invariant. The grammar for a variable declaration is:

$$\begin{array}{lll} varDecl & ::= & \texttt{VAR}\ vd^{+} \\ vd & ::= & ident,^{+}\ \mathbf{:}\ type\ [initialVal]\ ; \\ initialVal & ::= & \mathbf{:=}\ expr \end{array}$$

Exporting variables is not common in Modula-3 interfaces.

2.2.3 Private variables

It is often the case that the specification uses information that does not have to be accessible to the implementation.

To facilitate this, LM3 allows the declaration of *private variables*, which are notionally part of the state but which may be referenced only within specifications. These variables exist only within the specification domain and are associated with LSL sorts, not with types. For convenience, the obvious sorts are available to represent the common programming language types.

The grammar for private variables is:

```
privateVarDecl  ::=  <* PRIVATE ( ident,+ : sort; varSpec)+ *>
varSpec         ::=  [initial] [invar]
```

2.2.4 Types

Modula-3 has a rich space of types, including a notion of subtyping. All of the base types and the type constructors of the language are associated with LSL sorts in *LM3Trait*.

A type declaration may be annotated with a number of things. Not all will be appropriate in all cases. A fully annotated type declaration has the following form:

```
typeDecl     ::=  TYPE td+
td           ::=  ident [typeSpec] subtypeReln type ;
typeSpec     ::=  <* BASED ON [ident:]sort [initial] [invar] *>
subtypeReln  ::=  = | <:
type         ::=  ident | arrayType | recordType | ...
```

The first clause associates the type with a sort, which must be defined in one of the traits in the `USING` list. In particular, for any variable v of type T which is based on sort S, the value of v must be equal to a term of sort S for which T's invariant is true.

The *initial* clause, indicated by `INITIALLY`, introduces a predicate that must hold for initial values of a type. For a simple type this predicate constrains the initial values supplied in variable declarations. For an object type it is a constraint on the result of calls to `NEW`. Satisfaction of this constraint is an obligation of clients that use the type.

The *invar* clause, indicated by `INVARIANT`, introduces a predicate that is notionally conjoined to the pre- and post-condition of all procedures that may reference an element of the type. Such a procedure may assume the invariant on invocation and must guarantee it on exit. Object types, where invariants are inherited, are discussed separately in Section 2.3.3.

The *ident* represents a variable bound by universal quantification over the type.

Each type declared in the interface is associated with a sort. There are two categories of types:

1. types with modifiable values (`REF` types). That is T <: `REFANY` or `ROOT`. If such a T is `BASED ON` a sort S, the sort actually used to represent the type is SRef.

2. types with unmodifiable values (`VAL` types). In this case, there is no indirection and the sort is the one given in the `BASED ON`. Parameters of such a type are unmodifiable (and it is a checked error to try to modify a parameter of a `VAL` type). Modifications of such parameters are permitted only if they are declared as `VAR`. In this case the sort of the parameter is SVar which can be regarded as implicitly introducing an extra level of reference.

If we expand the Stack example to include the declaration of the type, we see:

```
INTERFACE Stack;
<* USING Stack(Real for E, RealStack for C) *>

TYPE T <* BASED ON RealStack *>
       <: REFANY;

(* declarations of exported  procedures - see below *)
END Stack.
```

For the example above, any variable of type Stack has sort RealStackRef. To get the value of such a variable, which has sort RealStack (which is produced by performing the given renaming on the trait Stack), we dereference the variable in the appropriate state using either `__\pre` or `__\post`.

2.2.5 Procedures

Modula-3 allows the use of a variety of constructs within procedures, including exceptions and threads. Therefore, procedure declarations may be annotated with a number of predicates. The meaning of a procedure specification is given as a predicate on a sequence of state pairs[2].

Atomic procedures

Most procedures written in Modula-3 are atomic. Since the specification of an atomic procedure is significantly simpler than in the non-atomic case, we consider this first.

If there are no exceptions and the procedure is atomic and unsynchronized, the grammar is:

procDecl	::=	`PROCEDURE` *ident* ([*signature*]) [: *type*] ; [<* *procSpec* *>]
signature	::=	{[*paramType*] *ident*,$^+$: *type* [*initialVal*] ;}$^+$
paramType	::=	`VALUE` | `VAR` | `READONLY`
procSpec	::=	[*globals*] [*privates*] [*letDecl*] [*prePred*] [*modifies*] *postPred*
globals	::=	((`WR` | `RD`) *ident* : *type*;)$^+$
privates	::=	`PRIVATE` varDecl$^+$
letDecl	::=	`LET` *let* `IN`
let	::=	*ident* `BE` *term*,$^+$
prePred	::=	`REQUIRES` *lm3Predicate*
modifies	::=	`MODIFIES` *term*,$^+$
postPred	::=	*atomicPostPred*
atomicPostPred	::=	`ENSURES` *lm3Predicate*

The components of the procedure specification are:

globals Declarations of all of the variables in the global state that are referenced by this procedure. For most purposes, this can be considered

[2]For a sequential language, this would be a relation on a single state pair. The introduction of concurrency means that we need to extend to a more complex semantic domain, where for each atomic action, a pair represents a state at the start of the action (*pre*) and one at the completion (*post*).

as an implicit extension to the parameter list. Global variables are annotated to indicate their intended use: globals may be indicated to be `WR` (writable) if they may be modified or `RD` (read-only) if it is not intended to change the value. It is an error to modify a global variable that has been declared to be `RD`.

privates Private variables local to each invocation.

letDecl Local shorthands defined by use of the `LET` construct. This is purely a syntactic substitution mechanism.

prePred a `REQUIRES` clause that defines the precondition of the procedure. This is a predicate that the caller must ensure is true in the state from which the procedure is invoked. If it is not , then nothing is guaranteed about the result, including termination. If this clause is omitted then it defaults to `true`, implying the procedure may be invoked from any state. The `REQUIRES` clause may reference only variables in the *pre* state.

modifies a `MODIFIES` clause that identifies the state components that the procedure is allowed to modify. If there is no `MODIFIES` clause, then the procedure may not modify anything. If the procedure is allowed to modify anything to which it has access, the shorthand is `MODIFIES ALL`. Modification must be consistent with the type of the parameter. For example, it is an error to mention a `VAL` or `READONLY` parameter in the `MODIFIES` list, remembering the default for M3 parameters is `VAL`. Only global variables declared as `WR` may be mentioned in the `MODIFIES` clause. This clause is a list of terms whose values are elements of a type that is modifiable (normally, a `REF`). From this list, a predicate asserting the validity of such modifications is derived.

postPred an `ENSURES` clause that gives the postcondition of the procedure. This is a predicate over the pair of states (the state on invocation and the state on termination) that must be true on exit from the specified procedure. Variables in the `ENSURES` clause may refer to values in both states and so are qualified with either `__\pre` or `__\post`. The unnamed return value of a procedure is represented by the pseudo variable `RESULT` which only has meaning in the final state and so must be qualified with `__\post`.

Formally, the meaning of such a specification of a procedure, when fully expanded, is given by the predicate :

$$prePred \Rightarrow (modPred \wedge postPred)$$

As an example, we add some functionality to our evolving Stack. Since we are following the example that can be found in the Modula-3 report, in which all procedures return a new Stack rather than modify the existing one, the specification is:

```
PROCEDURE Pop(VAR s:T): REAL;
<* REQUIRES NOT(isEmpty(s\pre\pre))
   MODIFIES s
   ENSURES s\post\post = pop(s\pre\pre)
       AND RESULT\post = top(s\pre\pre) *>
```

This tells us that this procedure should be called only when the value of s is not the empty stack, that the value of s may be modified and that the result and the final value of s will be the *top* and *pop* (from the *Stack* trait) of the initial value of s, respectively.

Since s is declared to be a `VAR` parameter, that implicitly adds a level of indirection. In other words, s is associated with the sort SRef rather than S. This explains the two uses of `__\pre` to get to a stack value: the first to dereference the `VAR` giving a SRef; the second to dereference this giving a S (remembering that T <: `REFANY`).

An aside

This specification of Stack is somewhat unusual. Since this has caused some confusion amongst some readers of this report, we'll discuss it a little.

More typically, one might expect the procedure to modify the existing Stack, rather than deliver a new one. To understand the contrast between these two styles, the specification of a Pop1 function in which the Stack is not a `VAR` parameter is:

```
PROCEDURE Pop1(s:T): REAL;
<* REQUIRES NOT(isEmpty(s\pre))
   MODIFIES s
   ENSURES s\post = pop(s\pre)
       AND RESULT\post = top(s\pre) *>
```

Here, the given Stack is changed. This is allowed since Stack is a `REF` type. We follow the former example in the rest of this report, since the extra level of indirection forces greater care in the specification and so serves our pedagogical purpose better, but feel we should point out that the strangeness is due to the desired behavior of the example rather than to the specification language.

Keyword predicates

Since the following example makes use of them, this is an appropriate place to mention LM3's keyword predicates. These keywords form an important part of the terms used to build predicates. Keyword predicates are :

`UNCHANGED`($v_1 \ldots v_n$), which asserts that the final values of these variables are equal to their initial values.

`FRESH`(*foo*), meaning that the storage assigned to *foo* is not shared with anything else in the state.

`CHECKEDRTE`, which is semantically equivalent to `true` but it warns the implementer that a checked run time error would occur

An interlude: the Stack

By this point, we have sufficient mechanism to complete our Stack example. Fitting together the pieces seen previously and adding the obvious procedures we get:

```
INTERFACE Stack;
<* USING Stack (REAL for E,  RealStack for C) *>

TYPE T <* BASED ON RealStack *>
       <: REFANY;

PROCEDURE Pop(VAR s:T): REAL;
<* REQUIRES NOT(isEmpty(s\pre\pre))
   MODIFIES s
   ENSURES s\post\post = pop(s\pre\pre)
       AND RESULT\post = top(s\pre\pre)
*>

PROCEDURE Push(VAR s: T;  x: REAL);
<* MODIFIES s
   ENSURES s\post\post = push(s\pre\pre, x)
*>

PROCEDURE Create(): T;
<* ENSURES isEmpty(RESULT\post) AND FRESH(RESULT) *>

END Stack.
```

This gives all the functional information that a client of this interface should ever need to know. In practice, the specification should be supplemented by textual comments, telling clients the (equally important) non-functional things they need to know.

Procedures with Exceptions

One of the common programming techniques in Modula-3 is the use of *exceptions* for abnormal termination of a procedure. The programming language allows you to declare the set of exceptions that may be raised by a procedure. The specification language permits you to describe an alternative result by using an `EXCEPT` clause, with guarded predicates that may be satisfied in place of the normal post condition. If any guards are true, then the procedure must satisfy the exception predicate of one of them, rather than the normal `ENSURES` predicate.

There is also the possibility of an exception being raised by a lower level of the program. The circumstances that lead to such an "abstraction failure" exception generally cannot be specified in terms of the state that is accessible to the caller. As far as the caller is concerned, these exceptions may be raised "arbitrarily". They are specified by `UNLESS` clauses. If a procedure specification has such clauses, then its final value may be either the value of one of the `UNLESS` predicates or that of the `ENSURES/EXCEPT` clause.

A state variable, `RAISE`, represents the value of a raised exception in the post state. This may take the value of any legitimate exception or the special value `RETURN` which indicates normal termination of the procedure. As a syntactic convenience, when an `EXCEPT` or an `UNLESS` clause is present, the term '`RAISE` = `RETURN`' is implicitly conjoined to the `ENSURES` clause.

The following extensions are made to the *procDecl* grammar:

procDecl	::=	. . . [*raisesList*] ; [*procSpec*]
raisesList	::=	`RAISES` { *ident*,* }
atomicPostPred	::=	`ENSURES` *lm3Predicate* [*except*] [*unless*]
except	::=	`EXCEPT` {*guardPredicate* ⇒ *exceptionPredicate*} |+
unless	::=	`UNLESS` *exceptionPredicate* |+

The keywords hopefully imply the correct interpretation of the postcondition, which is:

- the post predicate is true on exit
- **except** if any of the guards are true, then the corresponding exception predicate is true on exit. If more than one guard is true, a non-deterministic choice is allowed.
- **unless** one of the unguarded exception predicates is true on exit.

For example, for a procedure (loosely) specified as:

```
PROCEDURE x( ... );
<* REQUIRES prePred
   MODIFIES modPred
   ENSURES postPred
    EXCEPT g1 => ex1 | g2 => ex2
    UNLESS ex3 | ex4 *>
```

the meaning is :

$$prePred \Rightarrow (modPred \wedge$$
$$(\neg(g1 \vee g2) \wedge \; postPred \wedge \texttt{RAISE} = \texttt{RETURN})$$

$\vee(g1 \ \wedge \ ex1)$
$\vee(g2 \ \wedge \ ex2)$
$\vee \ ex3 \ \vee \ ex4)$

If, for example, our stack raised an exception on trying to push an object into a stack whose size was MaxSize, the procedure could be specified as:

```
EXCEPTION StackOverflow;

PROCEDURE Push(VAR s: T; x: REAL) RAISES{StackOverflow};
<* MODIFIES s
   ENSURES s\post\post = push(s\pre\pre, x)
    EXCEPT size(s\pre\pre) = MaxSize =>
                              RAISE\post = StackOverflow
                              AND UNCHANGED(s)
*>
```

Non-atomic procedures

The specification of non-atomic procedures is less well understood than that of atomic procedures. This area is a large part of our on-going research, and while the mechanisms proposed in this section are suitable for our current needs, it is likely that in the future, as the technology matures, we will adopt an approach based more closely on Lamport's Temporal Logic of Actions[9].

Post-conditions of atomic procedures range over exactly two states, a pre state and a post state. However, since Modula-3 allows concurrent threads of activity within an address space, this model is not sufficiently general for describing all Modula-3 procedures. In particular, we need to be able to describe intermediate states, which may be visible to other threads. To allow this, we specify a non-atomic procedure as being composed of a number of separate atomic actions. Each action is modeled as a relation on a pair of states, as before. The entire procedure can now be specified in terms of the sequence of state pairs, each pair representing one atomic action.

Some non-atomic procedures require a means of referring to the currently executing thread. This is designated by the keyword, `CURRENT`.

WHEN Predicate

Concurrency adds the need to specify *when* an action may take place, as well as *what* it does. This is given by a `WHEN` predicate. If a `WHEN` clause is given, the action is allowed only when the predicate is true in its pre state, which since there is concurrent activity may not be the same as the state at the call. The grammar is extended to include:

when	::=	`WHEN` *lm3Predicate*

as an optional component of a procedure specification.

Composite procedures

Explicit composition The simplest extension is to allow procedures that can be described as an *explicit* composition of subsidiary atomic actions. The grammar is extended to:

postPred	::=	*atomicPostPred* | *compositePostPred*
compositePostPred	::=	`COMPOSITION OF` *ident*;$^+$ `END` *action*$^+$
action	::=	`ACTION` *ident* [*when*] *atomicPostPred*

Examples of the use of composition can be found in Section 5.1.

Arbitrary composition

In this case, we describe the actions in much the same way, except that we do not know how many actions take place. We introduce a further notion of defining an action that may take place an arbitrary number of times[3]. Such an action is followed by the symbol $*$. As a syntactic convenience, we allow A; A* to be written as A+.

The full grammar for non-atomic procedures is therefore:

postPred	::=	*atomicPostPred* | *compositePostPred*
compositePostPred	::=	`COMPOSITION OF` *acts* ;$^+$ *action*$^+$ `END`
acts	::=	*ident*[$*$ | $+$] ; | (acts)

[3]We recognize that this extension is still not fully general. However, we appeal to a remark that Jim Horning attributes to Leslie Lamport, paraphrased as "90% of all procedures should appear atomic to their clients". We claim that the mechanism proposed here gives another 8% and we are cheerfully ignoring the remaining few for now.

This allows certain actions, say the first or the last, to be specifically constrained, while all other actions may be specified using the same predicate.

For example, the specification fragment:

```
COMPOSITION OF A+; B; C*; D END
```

tells us that this procedure can be modeled as: at least one A action, a B action, maybe some C actions and finally a D action.

2.3 Other Modula-3 features

Modula-3 is a modern language with some advanced features that require special attention in the specification language. Some, such as concurrent threads, have already been addressed above. The rest are gathered together into the following sections.

2.3.1 Procedure parameters

Modula-3 allows the programmer to pass procedures as parameters to other procedures. This feature is well understood at the program level since one can just call the passed procedure just like any other procedure. However, at the specification level, it is the procedure's specification that is of interest, not its implementation. "Calling" a specification has no meaning. The specification of a procedure is a predicate defining a relation over states, so passing p as a parameter, to a procedure R, actually means providing a predicate representing p. This predicate can then be used (with renaming of p's parameters) within the specification of R.

In order to be able to restrict the possible values of a procedure argument, we need to be able to talk about the specification of a formal procedure parameter in the `REQUIRES` clause. There is already a notation for giving the predicate specifying a procedure, that is `REQUIRES MODIFIES ENSURES`, which we can use. We extend the notion of a predicate to permit this form. This also allows us to place restrictions on a procedure type by using a `REQUIRES MODIFIES ENSURES` predicate in the type invariant.

Such a specification in the `REQUIRES` clause represents the weakest specification that any actual parameter has to meet. In reasoning about client code, the client uses the specification of the actual argument in place of the specification of the formal parameter.

We allow a specifier to refer to the components of a procedure P's specification, using P.`MODIFIES`, P.`REQUIRES` and P.`ENSURES`. Alternatively, the predicate representing the full specification is accessible as P.`SPEC`. In either case, renaming is permitted.

As an example, consider the specification of a 'parameterized' sorting routine, where the actual ordering function is given as a parameter.

We could specify this as:

```
<* USING TotalOrder(R for <) *>

PROCEDURE
 Sort(VAR data:  ARRAY OF INTEGER;
           ord:  PROCEDURE(a, b:INTEGER): BOOLEAN);
<* REQUIRES ord.SPEC IMPLIES
                 (MODIFIES NOTHING ENSURES RESULT\post = R(a, b))
        AND size(data\pre) > 1
   MODIFIES data
   ENSURES FORALL i,j \in inds(data\pre)
     ord.SPEC(data\post[i] for a,
              data\post[j] for b,
              i < j for RESULT\post)
     AND permutation(data\pre, data\post)
*>
```

This specification tells us that we are free to pass any function parameter whose specification is stronger than or equal to

$$(\texttt{MODIFIES NOTHING ENSURES RESULT\textbackslash post} = R(a, b))$$

or, in other words, has no side effects and implements some total order.

2.3.2 Intermediate states

In some circumstances, particularly if procedure parameters are involved (see Section 2.3.1), it is necessary to be able to refer to states that are not visible to the outside world, even in atomic procedures. To facilitate this, LM3 allows predicates that are explicitly quantified over states.

In such predicates, a state is bound by a quantifier, such as s is below. This state may be used in any place that either of the distinguished states, *pre* and *post*, may occur. The operator, `\eval`, which takes a state and a variable to a value. So, `v\pre` and `v\post` are equivalent to `\eval(pre, v)` and `\eval(post, v)` respectively. For convenience, `\eval(s, v)` may be written as `v\s`.

For example, if we have a specification of a procedure:

```
PROCEDURE double(n : INTEGER) : INTEGER;
<* REQUIRES n > 0
   ENSURES RESULT\post = 2 * n
*>
```

and a second procedure specified as :

```
PROCEDURE
  twice(p: PROCEDURE(n: INTEGER): INTEGER,
        i: INTEGER): INTEGER;
<* REQUIRES p.REQUIRES(i for n) AND
            FORALL s: State
                   (p.SPEC(s for post, i for n, j for RESULT\post)
                          IMPLIES p.REQUIRES(s for pre, j for n))
   ENSURES EXISTS  s: State
         (p.SPEC(s for post, i for n)
      AND p.SPEC(s for pre, RESULT\s for n)
*>
```

then we know that a call of `twice` with `double` as an actual value for p is allowed, since the specification of double satisfies the requirement on p. If we wanted to perform any reasoning about this call, then p.`REQUIRES`, for example, would be instantiated to the actual value of double.`REQUIRES`.

If we expand the predicates in such a call, we see for:

$$twice(double, 3)$$

the expanded predicate is:

$$\begin{aligned}
&\exists s : State \\
&\qquad (3 > 0 \Rightarrow \\
&\qquad\qquad \texttt{RESULT\textbackslash s} = 2 * 3 \wedge \\
&\qquad\qquad \texttt{RESULT\textbackslash s} > 0 \Rightarrow \texttt{RESULT\textbackslash post} = 2 * \texttt{RESULT\textbackslash s}
\end{aligned}$$

which indeed simplifies to **`RESULT\post`** $= 12$.

This example shows the need for quantification over states since we would be unable to refer to the intermediate result without such a mechanism.

2.3.3 Object types and methods

One of the ways in which Modula-3 differs from its predecessors in the Modula family is that it has subtyping with inheritance. LM3 has to support these features, which have not been addressed in previous Larch interface languages.

There is a problem. There are two distinct uses of inheritance within the Object Oriented community, only one of which represents true subtyping. It is reasonable to assume that if a type T1 is a subtype of T, then any properties we specify of T would still be required of T1. Unfortunately, Modula-3 can not enforce this semantic restriction, and it is often the case that programmers use inheritance simply to avoid rewriting some code, without really preserving subtyping. This would make it impossible to specify anything meaningful about the subtype relationship. LM3 supports only disciplined use of inheritance. Anything that is specified about a type must also be true for all subtypes (modulo appropriate rebinding of redefined operators). This will in certain cases require a programmer to perform actions (such as providing a new default method) that are not required by the programming language, per se, and certainly restricts the programmer's freedom to override methods arbitrarily.

An object value can be regarded as a record with an associated suite of procedures, called *methods*, giving operations bound to that record. An object type has a supertype and inherits both the structure and the default operations of this supertype. The LM3 keyword, `SELF`, refers to the current instantiation of the object.

The `INITIALLY` clause of an object type is a post-condition on any use of the function `NEW` with this type. Since this is not syntactically checkable, it is a proof obligation. The invariant on the type must be true on instance creation, and preserved by methods.

The specification of a subtype inherits the specifications of its supertype with its default methods and extends these specifications with specializations. Following the pattern of the Larch Shared Language, this inheritance is treated syntactically. Semantic interpretation is on the fully expanded form.

The grammar for an object type declaration in its simplest form, ignoring brands, traces, etc. (see Appendix B for the full detail) is:

objectTypeDecl	::=	*ancestor simpleObjectType*
ancestor	::=	*typeName* | ...
simpleObjectType	::=	OBJECT *fields* [*methodDecl*] END
methodDecl	::=	METHOD *method*$^+$
method	::=	*explicitMethod* | *strengthenMethodSpec*
explicitMethod	::=	*ident signature* [*defaultProc*] ; [*procSpec*]
strengthenMethodSpec	::=	<∗ STRENGTHEN *ident lm3Predicate* ∗>

This is most easily understood in terms of an example. We specify a somewhat artificial object called SetBag, which is the common ancestor of both Set and Bag. The traits defining the sorts and operators follow the interfaces.

```
INTERFACE SetBag;
<* USING SetBagTrait *>
IMPORT E FROM Element;

TYPE Space <: ROOT;
     T <* BASED ON t:SB
          INVARIANT nonNil(t) *>
        =  Space OBJECT
                 METHODS
                 insert(e: E) := insertDefault;
                 <* MODIFIES SELF
                    ENSURES SELF\post = add(SELF\pre, e)
                 *>

                 in(e: E): BOOLEAN := inDefault;
                 <* ENSURES RESULT\post = e \in SELF\pre *>
                 END;

PROCEDURE insertDefault(s: T; e: E);
<* MODIFIES s
   ENSURES s\post = add(s\pre, e)
*>

PROCEDURE inDefault(s: T; e: E): BOOLEAN;
<* ENSURES RESULT\post = e \in s\pre *>

END SetBag.
```

A minimal trait is given below. More realistic traits for Set and Bag are given in the LSL Handbook.

SetBagTrait: **trait**
introduces
$\{\} : \rightarrow$ SB
add: SB, Elem $\rightarrow$ SB
`__\in__` : SB, Elem $\rightarrow$ Bool
asserts
SB **generated by** ($\{\}$, add)
($\forall$ *s*: SB, *e*, *e1* : Elem)
$\neg$(*e* `\in` $\{\}$),
e `\in` add(*s*, *e1*) == (*e* = *e1*) | (*e* `\in` *s*)

We can then specify Set as a subtype:

```
INTERFACE Set;
<* USING SetTrait
   IMPORT SetBag;

TYPE T  <* BASED ON S *>
        =  SetBag.T OBJECT
                    METHODS
                    insert := setInsert;
                    END;

PROCEDURE setInsert(s: T; e: Elem)
<* MODIFIES s
   ENSURES s\post = add(s\pre, e);
END Set.
```

This interface uses the trait, SetTrait, which is:

SetTrait : **trait**
includes SetBagTrait(S **for** SB)
asserts S **partitioned by** (`\in`)

Note: the post condition of the insert procedure looks the same as in SetBag, but has a different meaning since Set.T is bound to a different sort from a different trait. Even if there were no explicit redefinition, we would still need to reinterpret the inherited predicates to ensure correct bindings of the operators.

For this to be a valid specification of a subtype, the following implications need to hold:

1. SetTrait $\Rightarrow$ SetBagTrait

2. setInsert.SPEC ⇒ SetBag.insert.SPEC

or, in other words, we must prove that Set is indeed a true subtype of SetBag.

We could define a second subtype of SetBag, namely Bag, which adds extra functionality.

```
INTERFACE Bag;
<* USING BagTrait *>
IMPORT SetBag;

TYPE T <* BASED  ON B *>
       =  SetBag.T OBJECT
                   METHODS
                   count(e: Elem):CARDINAL := countDefault;
                   <* ENSURES RESULT\post =
                              count(SELF\pre, e) *>
                   END;

PROCEDURE countDefault(b: T; e: Elem): CARDINAL;
<* ENSURES RESULT\post = count(b\pre, e) *>
END Bag.
```

The trait for this interface is:

BagTrait : **trait**
includes SetBagTrait(B **for** SB)
introduces count : B, Elem → Card
asserts B **partitioned by** count
  ∀ (b: B, e,$e1$: Elem)
  count({}, e) == 0
  count(add(b,e), $e1$) ==
    count(b,$e1$) + (**if** e = $e1$ **then** 1 **else** 0)
implies ∀(b : B, e,$e1$: Elem)
  count(b, e) > 0 ⇒e `\in` b

This completes the description of the constructs of LM3. A succinct summary of the syntax is given in Chapter 3.

Chapter 3

The Syntax of LM3

This section gives a complete description of the syntax of the LM3 language. Whilst much of this is paraphrased directly from the Modula-3 Report, it should be possible to follow the definition without knowledge of the grammar given there. The LM3 grammar is a superset of the Modula-3 interface grammar.

The grammar presented here is a collection of the components presented previously; it does not go down to the token level. A complete parsing grammar is given in Appendix B.

3.1 The LM3 reference grammar

interface	::=	`INTERFACE` *ident* ; [*traitUse*] *imports**
		[*intConstraints*] *declaration** `END` *ident* .
traitUse	::=	<∗ `USING` *traitRef,*$^+$ ∗>
imports	::=	[`FROM` *ident*] `IMPORT` *ident,*$^+$;
intConstraints	::=	<∗ initial ∗> | <∗ invar ∗> | <∗ initial invar ∗>
declaration	::=	*constDecl* | *varDecl* | *typeDecl* | *exceptionDecl*
		| *procDecl* | *privateVarDecl* | ...
initial	::=	`INITIALLY` *lm3Predicate*
invar	::=	`INVARIANT` *lm3Predicate*
constDecl	::=	*ident* [: *type*] = *constExpr*
varDecl	::=	`VAR` *vd*$^+$
vd	::=	*ident,*$^+$: *type* [*initialVal*] ;
initialVal	::=	:= *expr*
privateVarDecl	::=	<∗ `PRIVATE` (*ident,*$^+$: *sort*; varSpec)$^+$ ∗>
varSpec	::=	[*initial*] [*invar*]
typeDecl	::=	`TYPE` *td*$^+$
td	::=	*ident* [*typeSpec*] *subtypeReln* *type* ;
typeSpec	::=	<∗ `BASED ON` [*ident*:]*sort* [*initialPred*] [*invariantPred*] ∗>
subtypeReln	::=	= | <:
type	::=	*ident* | *arrayType* | *recordType* | ...

procDecl	::=	PROCEDURE *ident* ([*signature*]) [: *type*] [raisesList] ; [<* *procSpec* *>]
signature	::=	{[*paramType*] *ident*,$^+$: *type* [*initialVal*] ;}$^+$
paramType	::=	VALUE | VAR | READONLY
procSpec	::=	[*globals*] [*privates*] [*letDecl*] [*prePred*] [*modifies*] *postPred*
postPred	::=	*atomicPostPred*
globals	::=	((WR | RD) *ident* : *type*;)$^+$
privates	::=	PRIVATE varDecl$^+$
letDecl	::=	LET *let* IN
let	::=	*ident* BE *term*,$^+$
prePred	::=	REQUIRES *lm3Predicate*
modifies	::=	MODIFIES *term*,$^+$
raisesList	::=	RAISES { *ident*,* }
atomicPostPred	::=	ENSURES *lm3Predicate* [*except*] [*unless*]
except	::=	EXCEPT {*guardPredicate* ⇒ *exceptionPredicate*} |$^+$
unless	::=	UNLESS *exceptionPredicate* |$^+$
postPred	::=	*atomicPostPred* | *compositePostPred*
compositePostPred	::=	COMPOSITION OF *acts* ;$^+$ *action*$^+$ END
acts	::=	(*ident*[* | +];)|(*acts*)
action	::=	ACTION *ident* [*when*] *atomicPostPred*
when	::=	WHEN *lm3Predicate*

objectTypeDecl	::=	*ancestor simpleObjectType*
ancestor	::=	*typeName* | ...
simpleObjectType	::=	`OBJECT` *fields* [*methodDecl*] `END`
methodDecl	::=	`METHOD` *method*$^+$
method	::=	*explicitMethod* | <u>*strengthenMethodSpec*</u>
explicitMethod	::=	*ident signature* [*defaultProc*] `;` <u>[*procSpec*]</u>
<u>*strengthenMethodSpec*</u>	::=	<u><* `STRENGTHEN` *ident lm3Predicate* *></u>

Chapter 4

The semantics and checking of LM3

Thus far, most of the language has been described syntactically. In this chapter, we give the semantics underlying the syntax. The meaning of an interface specification is given by a translation into terms in the Larch Shared Language.

This translation, which is generated by the LM3 Checker, allows specifications to be mechanically type checked (or more accurately, sort checked for the sorts on which the types are based). Further checking, of the kind that requires more sophisticated tools (such as *LP*, the Larch Prover[5]) uses the same mechanism but is not addressed here.

This chapter presents a set of functions that translate the LM3 text into LSL, in a form that can be used for sort checking. We also give the LM3 traits. These are the traits associated with the primitive types and constructors of Modula-3.

4.1 The translation functions

This section presents the associated LSL declarations and terms for each construct in an LM3 interface specification. Each construct in the interface specification causes a corresponding phrase to be created in the trait, according to the following table. This presentation is not fully formal but indicates the translations performed by the LM3 checker.

For an interface Foo, by convention, we generate the trait in a file called FooTrait.lsl. Further, each trait implicitly imports *LM3Trait*.

For the components of an LM3 specification:

`INTERFACE` Foo	gives	*FooTrait:* **trait**
`USING` traitList	gives	**includes** *traitList*
`IMPORTS` impList	gives	**includes** *bazTrait* for each baz in impList
`CONST` x: T	gives	**introduces** $x: \to S$, where T is based on S
`VAR` x: T	gives	**introduces** $x: \to S$, where T is based on S
`EXCEPTION` e	gives	**introduces** $e: \to Except$
`TYPE` t `BASED ON` S (`REF` type)	gives	`__\pre,__\post`: SRef → S, `__\pre,__\post`: SRefVar → SRef, modifies, unchanged, fresh: SRef → Bool, modifies, unchanged: SRefVar → Bool
`TYPE` t `BASED ON` S (`VAL` type)	gives	`__\pre,__\post`: SVar → S, modifies, unchanged: SVar → Bool
`TYPE` t (M3 primitive)	gives	(see Section 4.2.1)
`TYPE` t (M3 constructor)	gives	(see Section 4.2.1)
`INVARIANT` predicate[1]	gives	predicate instantiated in the *pre* state, predicate instantiated in the *post* state
`PROCEDURE` p	gives	constant of the appropriate sort (for each formal in the parameter list, plus `RESULT` and `RAISE`)
`REQUIRES` predicate	gives	predicate as a term with each variable fully qualified with its sort
`MODIFIES` compList	gives	modifies(i: Sort) for each i in compList
`ENSURES` predicate	gives	a fully qualified term (according to the expansion of the `ENSURES` term given in Chapter 2)
`COMPOSITION` a_i ...	gives	isAction(a_i)
`ACTION` a	gives	a:→ Action, expand body as for procedure

To illustrate the translation, the Stack example is translated to the following which then sort checks correctly using the LSL checker:

```
% A simple example of an interface and its translation
```

[1]Remember that this translation is for sort checking only. For full semantic checking, the invariant predicate would be conjoined to the pre- and post-conditions of any procedure whose formals or globals contain variables of this type.

```
% The example is a little perverse since the procedures insist
% on returning new stacks rather than modifying the ones they have.
% This is the way the interface is specified in the report and it's
% a good example here since the extra level of indirection forces
% one to be more careful! In this example, repeated declarations
% (e.g. s: -> RealStackRefVar) are given only once.

% '%' lines give the LM3 specification. The lines following are
% the LSL translation.

% INTERFACE Stack;
StackTrait : trait

% USING Stack(REAL for E, RealStack for C)
includes Stack(Real for E, RealStack for C)

introduces
% TYPE T <* BASED ON RealStack *>
%       <: REFANY;
modifies, unchanged, fresh: RealStackRef -> Bool %since T <: REFANY
modifies, unchanged: RealStackRefVar -> Bool  %for VAR parameters
__\pre, __\post: RealStackRef -> RealStack
__\pre, __\post: RealStackRefVar -> RealStackRef

% PROCEDURE Pop(VAR s:T): REAL;
s: -> RealStackRefVar
RESULT: -> Real
% PROCEDURE Push(VAR s:T; x: REAL);
x: -> Real
% PROCEDURE Create():T;
RESULT: -> RealStackRef

asserts equations
% PROCEDURE Pop(VAR s:T): REAL;
% REQUIRES NOT(isEmpty(s\pre\pre))
~isEmpty(s:RealStackRefVar\pre\pre)           \;
%    MODIFIES s
modifies(s: RealStackRefVar)                  \;
%    ENSURES s\post\post = pop(s\pre\pre) AND FRESH(s\post)
%            AND RESULT = top(s\pre\pre)
(s:RealStackRefVar\post\post) = pop(s:RealStackRefVar\pre\pre) &
fresh(s:RealStackRefVar\post) &
RESULT: Real = top(s:RealStackRefVar\pre\pre) \;
```

```
% PROCEDURE Push(VAR s:T; x: REAL);
% MODIFIES s
modifies(s: RealStackRefVar)                    \;
%    ENSURES s\post\post = push(s\pre\pre,x)
%            AND FRESH(s\post)
s:RealStackRefVar\post\post =
                      push(s:RealStackRefVar\pre\pre, x: Real) &
fresh(s:RealStackRefVar\post)                   \;
%
% PROCEDURE Create():T;
% ENSURES RESULT\post = new AND FRESH(RESULT\post)
RESULT:RealStackRef\post = new: RealStack &
fresh(RESULT:RealStackRef)
% END Stack.
```

4.2 The LM3Trait

For any user-defined types, the corresponding sort is given in the type specification in the interface. There is no actual interface giving the definitions for the built-in types of Modula-3. The correspondence of such types with sorts is actually built into the LM3 Checker, but for pedagogical reasons, we present the specifications that we would associate with the built-in types if such an interface existed.

4.2.1 The interfaces

For simple types:

```
TYPE
        INTEGER    <* BASED ON i: Int
                      INVARIANT MinInt <= i <= MaxInt *>;
        CARDINAL   <* BASED ON c: Int
                      INVARIANT 0 <= c <= MaxInt *>;
        BOOLEAN    <* BASED ON b: Bool *>;
        CHAR       <* BASED ON c: Char *>;
        REAL       <* BASED ON r: Float *>;
        REFANY     <* BASED ON r: RefAny *>;
        TEXT       <* BASED ON t: Text
                      INITIALLY t = empty *>;
```

```
MUTEX      <* BASED ON m: Mu
              INITIALLY holder(m) = none *>;
THREAD     <* BASED ON Th *>;
```

For type constructors, the notional interface is actually a schema, since each use of a constructor needs an appropriately instantiated sort. In some cases, we can use the LSL shorthands to produce an appropriate trait.

In the following, we generate traits according to the indicated instantiation.

```
ARRAY OF X             <* BASED ON Array(X, XArray) *>;
SET OF X               <* BASED ON Set(X, XSet) *>;
REF X                  <* BASED ON RefSort(X, XRef) *>;
RECORD ...             <* BASED ON tuple of ... *>;
any enumeration {...} <* BASED ON enumeration of ... *>;
```

We made a deliberate choice *not* to provide a default sort for the object constructor. Since objects are by their very nature a representation of an abstraction, the specifier will always provide the trait that represents that abstraction.

Procedure types also have a special interpretation. For any procedure parameter, we generate a trait which provides the operators `.SPEC`, `.REQUIRES`, `.MODIFIES` and `.ENSURES` which deliver Boolean results. These operators are placeholders and are substituted by the predicates of the specification of the actual parameter in any reasoning about a use of the interface. We adopt a convention by which the use of `p.REQUIRES` represents `p.REQUIRES`(*pre*, *post*, $p_1, \ldots, p_n, p_{\texttt{RESULT}}$). This gives default values for the parameters and allows renaming using the mechanism from LSL. The trait is generated by the checker, following a syntactic template.

For example, for p : `PROCEDURE(INTEGER):INTEGER`, we generate a trait of the form:

ProcedureIntegerToInteger : **trait**

introduces

__.SPEC, __.REQUIRES, __.MODIFIES, __.ENSURES

: $PIntToInt, st, st, Int, Int \rightarrow Bool$

asserts

$\forall p : PIntToInt, pre, post : State, i, j : Int$

$$
\begin{aligned}
&p.SPEC(pre, post, i, j) == \\
&\quad (p.REQUIRES(pre, post, i, j) \Rightarrow \\
&\qquad p.MODIFIES(pre, post, i, j) \wedge \\
&\qquad p.ENSURES(pre, post, i, j))
\end{aligned}
$$

4.2.2 The trait

LM3Trait is constructed by including the base traits, plus an instantiation of a template for each use of a constructor.

LM3Trait : **trait**
includes *Integer*, *Boolean*, *Character*, *Float*, *Text*, *Thread*
plus instantiations of *Set*, *Ref*, *Array*, *Enumeration*, *Procedure* **for each use in an interface**

The included traits are given in Appendix A.

Chapter 5

Examples

In this chapter, we present two examples of LM3 specifications. The first presents the specification of Threads, which is one of the required interfaces for the Modula-3 implementation. Being a low-level interface this is somewhat atypical but illustrates the specification of non-atomic procedures. The second example presents a complete interface providing the functionality of I/O streams.

5.1 The Threads interface

This example was originally presented by Birrell et al. in [1] together with an accompanying discussion on programming with Threads. This section is a translation of that paper to the current LM3 and much of the text and all of the credit is due to the original authors. We present the formal specification without much commentary. This specification is self-contained; none of the informal description of threads is needed to understand its precise semantics. However, it is intended to be used in conjunction with informal material, such as that in [2]. The informal material provides intuition and says how the primitives are intended to be used.

The traits used by this interface can be found in Appendix A.

5.1.1 Mutex, Acquire, Release

```
<* USES Mutex *>
TYPE Mutex
   <* Mutex BASED ON m: Mu INITIALLY holder(m) = none *>
   <: ROOT;

PROCEDURE Acquire(VAR m: Mutex);
<* MODIFIES m
   WHEN holder(m\pre) = none
   ENSURES holder(m\post) = CURRENT *>

PROCEDURE Release(VAR m: Mutex);
<* REQUIRES holder(m\pre) = CURRENT
   MODIFIES m
   ENSURES holder(m\post) = none *>
```

If `Release(m)` is executed when there are several threads waiting to perform `Acquire(m)`, the `WHEN` clause of each of them will be satisfied. Only one thread will hold `m` next, because—by atomicity of `Acquire`—it must appear that one of the `Acquires` is executed first; its `ENSURES` clause falsifies the `WHEN` clauses of all the others. Our specification does not say which of the blocked threads will be unblocked first, nor when this will happen.

5.1.2 Semaphore, P, V

```
<* USES Semaphore *>
TYPE Semaphore
    <* BASED ON s: Semaphore
       INITIALLY s = unlocked *>
    = { unlocked, locked };

PROCEDURE P(VAR s: Semaphore);
<* MODIFIES s
   WHEN s\pre = unlocked
   ENSURES s\post = locked *>

PROCEDURE V(VAR s: Semaphore);
<* MODIFIES s
   ENSURES s\post = unlocked *>
```

5.1.3 Blocking and unblocking on condition variables

```
<* USES Thread, Set(Th, ThreadSet) *>
TYPE Condition
    <* BASED ON c: ThreadSet
       INITIALLY c = {} *>
    <: ROOT;

PROCEDURE Wait(VAR m: Mutex; VAR c: Condition);
<* REQUIRES holder(m) = CURRENT
   MODIFIES m, c
   COMPOSITION OF Enqueue; Resume END
   ACTION Enqueue
         ENSURES holder(m\post) = none
         AND c\post = c \union {CURRENT}
   ACTION Resume
         WHEN holder(m\pre) = none AND NOT(CURRENT \in c\pre)
         ENSURES holder(m\post) = CURRENT AND UNCHANGED(c) *>

PROCEDURE Signal(VAR c: Condition);
<* MODIFIES c
   ENSURES c\post = {} OR c\post \subset c *>

PROCEDURE Broadcast(VAR c: Condition);
<* MODIFIES c
   ENSURES c\post = {} *>
```

Any implementation that satisfies `Broadcast`'s specification also satisfies `Signal`'s. We cannot strengthen `Signal`'s postcondition: the recommended implementation of `Signal` usually unblocks just one waiting thread, but may unblock more.

5.1.4 Alerts

```
<* USES Thread *>
EXCEPTION Alerted;

PROCEDURE Alert(t: Thread);
<* MODIFIES t
   ENSURES t.alertPending\post *>

PROCEDURE TestAlert(): BOOLEAN;
<* MODIFIES CURRENT
   ENSURES IF RESULT\post
           THEN CURRENT.alertPending\pre AND
                NOT(CURRENT.alertPending\post)
           ELSE UNCHANGED(CURRENT) *>

PROCEDURE AlertP(VAR s: Semaphore) RAISES {Alerted};
<* MODIFIES s, CURRENT
   WHEN s\pre = unlocked OR CURRENT.alertPending\pre
   ENSURES s\post = locked AND
           UNCHANGED(CURRENT.alertPending)
    UNLESS RAISE = Alerted AND CURRENT.alertPending\pre
           AND NOT(CURRENT.alertPending\post) AND UNCHANGED(s) *>
```

The `UNLESS` clause in `AlertP` allows non-determinism.

```
PROCEDURE AlertWait(VAR m: Mutex; VAR c: Condition)
                   RAISES {Alerted};
<* REQUIRES holder(m\pre) = CURRENT
   MODIFIES m, c, CURRENT
   PRIVATE alertChosen: BOOLEAN
   COMPOSITION OF Enqueue; ChooseOutcome; GetMutex END
   ACTION Enqueue
   ENSURES holder(m\post) = none AND c\post = c \union
                     {CURRENT}
                 AND UNCHANGED(CURRENT)
   ACTION ChooseOutcome
   WHEN NOT(CURRENT \in c\pre) OR CURRENT.alertPending\pre
   ENSURES alertChosen\post = CURRENT \in c\pre
    AND UNCHANGED(holder(m))
    AND c\post = delete(CURRENT, c\pre)
    AND CURRENT.alertPending\post = (CURRENT.alertPending\post
             AND NOT(alertChosen))
   ACTION GetMutex
   WHEN holder(m\pre) = none
   ENSURES NOT(alertChosen\pre) AND holder(m\post) = CURRENT
           AND UNCHANGED(CURRENT)
    UNLESS RAISE = Alerted AND alertChosen\pre
           AND holder(m\post) = CURRENT) AND UNCHANGED(c)
           AND UNCHANGED(CURRENT) *>
```

5.2 The IO Streams interface

Finally, we present a definition of the IO Stream interface that forms part of the standard IO package used at the Systems Research Center. The interface is taken from Brown & Nelson [3].

5.2.1 An IOStreams package

The package makes use of the *partially opaque types* of Modula-3 to present a safe and efficient IO package. The report describes a number of types ranging from the abstract readers and writers, down to machine dependent unsafe modules that exploit low-level features to achieve efficiency. The reader is referred to [3], both for further detail and for a good example of well structured Modula-3 programing.

We address the input classes and present the most abstract reader and a more concrete realization of it. We borrow enough of explanation from [3] to make the interface comprehensible.

5.2.2 The Rd interface

Rd.T, pronounced reader, is a type that provides functions for accessing a character input stream. Abstractly, it consists of:

len the number of source characters

src a sequence of characters

cur an integer index into src, representing the current position

avail an integer representing the number of characters available

closed a boolean that's true for a Rd that has been closed

seekable a boolean that's true if the current position can be set to anywhere in src

intermittent a boolean that's true if the source is available in increments rather than all at once.

Since there are many concrete representations of readers that may fail in any number of different ways, the abstract class declares an exception `Failure` which takes a `REFANY` and is used to represent all failures.

The following is the abstract Rd interface, with some uninteresting functions omitted.

Rd.i3

```
INTERFACE  Rd; <* USING  Reader *>
FROM  Thread IMPORT  Alerted;

TYPE  T  <* BASED  ON  rd: R
            INVARIANT
                intermittent(rd) OR avail(rd) = len(rd) + 1 *>
        <: ROOT;

     Code = {Closed, Unseekable, Intermittent, CantUnget};

EXCEPTION  EndOfFile;
           Failure(REFANY);
           Error(Code);


PROCEDURE  GetChar(rd: T): CHAR
           RAISES {EndOfFile, Failure, Alerted, Error};
(* Return the next character from the src of rd *)
    <* MODIFIES  rd
       WHEN  avail(rd\pre) > cur(rd\pre)
       ENSURES  RESULT\post  =  currentVal(rd\pre)
           AND rd\post = setCur(rd\pre, cur(rd\pre)+1)
           AND canUnget(rd\post)
       EXCEPT  closed(rd\pre)          => RAISE  = Error(Closed)
                                          AND UNCHANGED(rd)
            | cur(rd\pre)= len(rd\pre) => RAISE  = EndOfFile
                                          AND UNCHANGED(rd)
            | isNil(rd\pre)            => CHECKEDRTE
       UNLESS RAISE  = Failure(x)
            | RAISE  = Alerted *>


PROCEDURE  EOF(rd: T): BOOLEAN  RAISES {Failure,Alerted,Error};
(* Return true iff rd is at end-of-file *)
    <* WHEN  avail(rd\pre) > cur(rd\pre)
       ENSURES  RESULT\post  = (cur(rd\pre) = len(rd\pre))
       EXCEPT  closed(rd\pre) => RAISE  = Error(Closed)
            |  isNil(rd\pre)  => CHECKEDRTE
       UNLESS RAISE  = Failure(x)
            | RAISE  = Alerted *>
```

```
PROCEDURE  UnGetChar(rd: T) RAISES {Error};
(* Push back the last char read,
   so the next call to GetChar will read it again *)
    <* REQUIRES  cur(rd\pre) > 0
       MODIFIES rd
       ENSURES  rd\post
                 = setCanUnget(setCur(rd\pre, cur(rd\pre)-1), false)
       EXCEPT closed(rd\pre) => RAISE  = Error(Closed)
                                AND UNCHANGED(rd)
            | isNil(rd\pre)  => CHECKEDRTE
       UNLESS  RAISE  = Error(CantUnget)
               AND NOT(canUnget(rd\pre)) *>


PROCEDURE  CharsReady(rd: T): CARDINAL  RAISES {Failure, Error};
(* Return some number of chars that can be read
   without indefinite waiting *)
    <* ENSURES  IF avail(rd\pre) = cur(rd\pre)
                THEN RESULT\post  = 0
                ELSE (1 \leq RESULT\post
                      AND RESULT\post
                          \leq (avail(rd\pre) - cur(rd\pre)))
       EXCEPT  closed(rd\pre) => RAISE  = Error(Closed)
            |  isNil(rd\pre)   => CHECKEDRTE
       UNLESS  RAISE  = Failure(x) *>



PROCEDURE  GetText(rd: T; len: INTEGER): TEXT
          RAISES  {Failure, Alerted, Error};
(* Get chars from rd until exhausted or len chars have been read *)
    <* MODIFIES  rd
       LET  inc = MIN(len, len(rd\pre) - cur(rd\pre)) IN
       ENSURES (RESULT\post  =
                      FromStr(subSrc(rd\pre, cur(rd\post))))
               AND (rd\post
                    = setCanUnget(
                          setCur(rd\pre, cur(rd\pre)+ inc),
                                                     inc > 0))
       EXCEPT closed(rd\pre) => RAISE  = Error(Closed)
            | isNil(rd\pre)  => CHECKEDRTE
       UNLESS RAISE  = Failure(x)
            | RAISE  = Alerted *>
```

```
PROCEDURE  GetLine(rd: T): TEXT
          RAISES {EndOfFile, Failure, Alerted, Error};
(* Read chars until newline or rd is exhausted *)
    <* MODIFIES rd
       ENSURES RESULT\post  = FromStr(subSrc(rd\pre, cur(rd\post))
               AND ((cur(rd\post) = len(rd\pre))
                   OR isLine(RESULT\post & NewLine))
               AND rd\post =
                   setCanUnget(setCur(rd\pre, cur(rd\post)), true)
       EXCEPT  cur(rd\pre)= len(rd\pre) => RAISE  = EndOfFile
            |  closed(rd\pre)           => RAISE  = Error(Closed)
            |  isNil(rd\pre)            => CHECKEDRTE
       UNLESS RAISE  = Failure(x)
            | RAISE  = Alerted *>


PROCEDURE  GetIndex(rd: T): CARDINAL  RAISES {Error};
(* Return the current index *)
    <* ENSURES  RESULT\post  =  cur(rd\pre)
       EXCEPT  closed(rd\pre) => RAISE  = Error(Closed)
            | isNil(rd\pre)   => CHECKEDRTE  *>


PROCEDURE  GetLength(rd: T): CARDINAL
          RAISES {Failure, Alerted, Error};
(* Return the length of the src *)
    <* ENSURES  RESULT\post  =  len(rd\pre)
       EXCEPT closed(rd\pre)       => RAISE  = Error(Closed)
            | intermittent(rd\pre) => RAISE  = Error(Intermittent)
            | isNil(rd\pre)        => CHECKEDRTE
       UNLESS  RAISE  = Failure(x)
            | RAISE  = Alerted *>


PROCEDURE  Seek(rd: T; n: CARDINAL)
          RAISES {Failure, Alerted, Error};
(* Set cur to n *)
    <* MODIFIES  rd
       ENSURES  rd\post =
                 setCanUnget(
                      setCur(rd\pre, MIN(n, len(rd\pre)), false)
       EXCEPT closed(rd\pre)        => RAISE  = Error(Closed)
                                       AND UNCHANGED(rd)
```

```
            | NOT(seekable(rd\pre)) => RAISE  = Error(Unseekable)
                                       AND UNCHANGED(rd)
            | isNil(rd\pre)         => CHECKEDRTE
       UNLESS  RAISE  = Failure(x)
            | RAISE  = Alerted *>


PROCEDURE  Close(rd: T) RAISES {Failure, Alerted};
(* Close rd *)
    <* MODIFIES  rd
       ENSURES  rd\post = close(rd\pre)
       EXCEPT  isNil(rd\pre) => CHECKEDRTE
       UNLESS RAISE  = Failure(x) AND closed(rd\post)
            | RAISE  = Alerted AND closed(rd\post) *>




PROCEDURE  Intermittent(rd: T): BOOLEAN  RAISES {};
(* Return true if rd is intermittent *)
    <* ENSURES  RESULT\post  =  intermittent(rd\pre)
       EXCEPT  isNil(rd\pre) => CHECKEDRTE  *>


PROCEDURE  Seekable(rd: T): BOOLEAN  RAISES {};
(* Return true if rd is seekable *)
    <* ENSURES  RESULT\post  = seekable(rd\pre)
       EXCEPT  isNil(rd\pre) => CHECKEDRTE  *>


PROCEDURE  Closed(rd: T): BOOLEAN  RAISES {};
(* Return true if rd is closed *)
    <* ENSURES  RESULT\post  = isClosed(close(rd\pre))
       EXCEPT  isNil(rd\pre) => CHECKEDRTE  *>
END  Rd.
```

Reader.lsl

The Rd interface is based on a trait that defines the basic operations on an Rd.T.

Reader : **trait**
  includes *Char*, *Natural*, *Sequence*(*Nat*, *Char*, *CharSeq*, *Nat* **for** *Card*),
    Text(*CharSeq*), *Integer*(*Nat* **for** *Int*)
  RT **tuple of** *src* : *CharSeq*, *cur* : *Nat*, *avail* : *Nat*, *closed* : *Bool*,
    seekable : *Bool*, *intermittent* : *Bool*, *canUnget* : *Bool*
  introduces
    $new :\rightarrow R$
    $appendSrc : R, CharSeq \rightarrow R$
    $close : R \rightarrow R$
    $setAvail : R, Nat \rightarrow R$
    $setCanUnget : R, Bool \rightarrow R$
    $setCur : R, Nat \rightarrow R$
    $avail : R \rightarrow Nat$
    $canUnget : R \rightarrow Bool$
    $closed : R \rightarrow Bool$
    $cur : R \rightarrow Nat$
    $currentVal : R \rightarrow Char$
    $intermittent : R \rightarrow Bool$
    $isNil : R \rightarrow Bool$
    $len : R \rightarrow Nat$
    $seekable : R \rightarrow Bool$
    $src : R \rightarrow CharSeq$
    $subSrc : R, Nat \rightarrow Text$
    $proj : R \rightarrow RT$
  asserts
    $\forall\ r : R, n : Nat, b : Bool, cs : CharSeq$
      $proj(appendSrc(r, cs)) == set\_src(proj(r), proj(r).src \| cs)$
      $proj(close(r)).closed$
      $proj(setAvail(r, n)) == set\_avail(proj(r), n)$
      $proj(setCanUnget(r, b)) == set\_canUnget(proj(r), b)$
      $proj(setCur(r, n)) == set\_cur(proj(r), n)$
      $avail(r) == proj(r).avail$
      $canUnget(r) == proj(r).canUnget$
      $closed(r) == proj(r).closed$
      $cur(r) == proj(r).cur$
      $currentVal(r) == (proj(r).src)[proj(r).cur]$
      $intermittent(r) == proj(r).intermittent$
      $isNil(r) == r = new$
      $len(r) == size(proj(r).src)$

$seekable(r) == proj(r).seekable$
$src(r) == proj(r).src$
$subSrc(r, n) ==$
$\quad fromString(subsequence(proj(r).src,$
$\quad proj(r).cur,$
$\quad proj(r).cur - n))$

5.2.3 The RdClass interface

This interface represents a realization of the Rd abstraction, showing some of the implementation detail. This interface illustrates the use of inheritance and revelation within LM3 specifications.

RdClass.i3

RdClass reveals that every reader contains a buffer of characters. The variable `buff`, together with `st`, `hi` and `lo` represent a part of `src`. The invariant describes the relationship between the representation and the abstraction given in the supertype. `Private` is an opaque type that allows the hiding of further implementation detail that is not relevant at this level.

```
INTERFACE RdClass <* USING ReaderClass *>;
IMPORT Rd;
FROM Thread IMPORT Alerted;
FROM Rd IMPORT Failure, Error;

TYPE Private  <* BASED ON PS *>
        <: ROOT;
     SeekResult = {Ready, WouldBlock, Eof};

REVEAL
        Rd.T <* BASED ON rd: RC
                INVARIANT
                   FORALL i \in {lo(rd) .. hi(rd)}
                      (buff(rd)[st(rd) + i - lo(rd)] = src(rd)[i])
                    AND cur(rd) = MIN(concreteCur(rd), len(rd))
                    AND NOT (intermittent(rd) AND seekable(rd)) *>
           = Private BRANDED OBJECT
             buff: REF ARRAY OF CHAR;
             st: CARDINAL; (* index into buff *)
             lo, hi, cur : CARDINAL; (* indexes into src(rd)*)
             closed, seekable, intermittent: BOOLEAN;
             METHODS
```

```
            seek(dontBlock: BOOLEAN): SeekResult
               RAISES {Failure, Alerted, Error};
               <* REQUIRES seekable(SELF\pre)
                        OR concreteCur(SELF\pre) = hi(SELF\pre)
                  MODIFIES SELF
                  ENSURES RESULT\post = Ready
                      AND cur(SELF\post) = concreteCur(SELF\post)
                       OR RESULT\post = Eof AND
                          cur(SELF\pre) = concreteCur(SELF\pre)
                            AND cur(SELF\pre) = len(SELF\pre)
                       OR RESULT\post = WouldBlock AND
                          dontBlock AND
                          avail(SELF\pre) = cur(SELF\pre)
                   UNLESS RAISE = Failure(x)
                        | RAISE = Alerted *>
            length(): CARDINAL RAISES {Failure, Alerted, Error}
               <* ENSURES RESULT\post  =  len(SELF\pre)
                  EXCEPT closed(SELF\pre)     =>
                               RAISE = Error(Closed)
                       | intermittent(SELF\pre) =>
                               RAISE  = Error(Intermittent)
                       | isNil(SELF\pre)        => CHECKEDRTE
                  UNLESS RAISE  = Failure(x)
                       | RAISE  = Alerted *>
               := LengthDefault;
            close() RAISES {Failure, Alerted, Error}
               <* MODIFIES  SELF
                  ENSURES  SELF\post = close(SELF\pre)
                  EXCEPT  isNil(SELF\pre) => CHECKEDRTE
                  UNLESS  RAISE  = Failure(x)
                          AND closed(SELF\post)
                       |  RAISE  = Alerted
                          AND closed(SELF\post)*>
               := CloseDefault;
END;

PROCEDURE Lock(rd: Rd.T) RAISES {};
(* Lock rd *)
<* REQUIRES NOT(locked(rd\pre))
   MODIFIES rd
   ENSURES locked(rd\post) *>

PROCEDURE LengthDefault(rd: Rd.T): CARDINAL
          RAISES {Failure, Alerted, Error}
```

```
(* A dummy default - must be overridden for any actual rd *)
<* ENSURES CHECKEDRTE
   EXCEPT closed(rd\pre)     => RAISE  = Error(Closed)
        |  intermittent(rd\pre) => RAISE  = Error(Intermittent)
   UNLESS RAISE  = Failure(x)
        |  RAISE  = Alerted *>

PROCEDURE CloseDefault(rd: Rd.T): CARDINAL
          RAISES {Failure, Alerted, Error}
(* Close rd and set the buffer to NIL *)
<* ENSURES buff(rd\post) = new AND rd\post = close(rd\pre)
   EXCEPT isNil(rd\pre) => CHECKEDRTE
   UNLESS RAISE  = Failure(x) AND  closed(rd\post)
        |  RAISE  = Alerted AND  closed(rd\post) *>

END RdClass.
```

ReaderClass.lsl

The trait for RdClass has much in common with Reader, adding extra components to represent the additional fields of Rd.

ReaderClass : **trait**
  includes *Reader*, *Reader*(*RC* **for** *R*)
  RCT **tuple of** *read* : *R*, *buff* : *CharSeq*, *st* : *Nat*, *lo* : *Nat*, *hi* : *Nat*, *concreteCur* : *Nat*, *locked* : *Bool*
  introduces
    $new : \to RC$
    $buff : RC \to CharSeq$
    $st : RC \to Nat$
    $lo : RC \to Nat$
    $hi : RC \to Nat$
    $concreteCur : RC \to Nat$
    $src : RC \to CharSeq$
    $cur : RC \to Nat$
    $len : RC \to Nat$
    $intermittent : RC \to Bool$
    $seekable : RC \to Bool$
    $avail : RC \to Nat$
    $closed : RC \to Bool$
    $isNil : RC \to Bool$
    $locked : RC \to Bool$
    $lock : RC \to RC$

$close : RC \rightarrow RC$
$proj : RC \rightarrow RCT$
$reader : RC \rightarrow R$

asserts

$\forall\ rc : RC$

$reader(rc) == proj(rc).read$
$buff(rc) == proj(rc).buff$
$st(rc) == proj(rc).st$
$lo(rc) == proj(rc).lo$
$hi(rc) == proj(rc).hi$
$concreteCur(rc) == proj(rc).concreteCur$
$locked(rc) == proj(rc).locked$
$locked(lock(rc))$
$src(rc) == src(reader(rc))$
$cur(rc) == cur(reader(rc))$
$len(rc) == len(reader(rc))$
$intermittent(rc) == intermittent(reader(rc))$
$seekable(rc) == seekable(reader(rc))$
$avail(rc) == avail(reader(rc))$
$closed(rc) == closed(reader(rc))$
$isNil(rc) == rc = new$
$proj(close(rc)).read == close(reader(rc))$

Acknowledgments

Many people have provided assistance with the work that is described in this report. I would like to specifically mention the following.

Jim Horning was involved in all stages of the design of LM3 and has also provided many valuable comments on various drafts of this report. The Larch group, particularly John Guttag and Jeannette Wing, influenced LM3 by discussion and comments throughout its design. Luca Cardelli, Martín Abadi and Cynthia Hibbard made helpful comments on draft versions of the report. Greg Nelson read and commented on early versions of the IO Streams specification. Joe Wild, Gary Feldman, Bill McKeeman and Steve Garland provided the tools used to check and process some of the specifications and grammars given here.

Appendix A

Traits

This appendix contains most of the standard traits referenced in this report. A few traits, such as Float, are still under development and are not ready for inclusion here. Complete versions of these will be given in [8].

A.1 Boolean

Boolean : **trait**

% This trait is given for documentation only.

% It is implicit in LSL.

introduces

$true, false : \rightarrow Bool$

$\neg \_\_ : Bool \rightarrow Bool$

$\_\_ \wedge \_\_, \_\_ \vee \_\_, \_\_ \Rightarrow \_\_ : Bool, Bool \rightarrow Bool$

asserts

Bool **generated by** *true*, *false*

$\forall\ b : Bool$

$\neg true == false$

$\neg false == true$

$true \wedge b == b$

$false \wedge b == false$

$true \vee b == true$

$false \vee b == b$

$true \Rightarrow b == b$

$false \Rightarrow b == true$

implies

$AC(\wedge, Bool)$,
$AC(\vee, Bool)$,
$Distributive(\vee$ **for** $+, \wedge$ **for** $*, Bool$ **for** $T)$,
$Distributive(\wedge$ **for** $+, \vee$ **for** $*, Bool$ **for** $T)$,
$Involutive(\neg\_\_, Bool)$,
$Transitive(\Rightarrow$ **for** $\diamond, Bool$ **for** $T)$
$\forall\ b_1, b_2, b_3 : Bool$
$\neg(b_1 \wedge b_2) == \neg b_1 \vee \neg b_2$
$\neg(b_1 \vee b_2) == \neg b_1 \wedge \neg b_2$
$b_1 \vee (b_1 \wedge b_2) == b_1$
$b_1 \wedge (b_1 \vee b_2) == b_1$
$b_2 \vee \neg b_2$
$(b_1 = b_2) \vee (b_1 = b_3) \vee (b_2 = b_3)$
$b_1 \Rightarrow b_2 == \neg b_1 \vee b_2$

A.2 Char

Char: **trait**
Ch **enumeration of** `\000, ... \377`

A.3 Float

Float: **trait**
includes *Integer*, *DerivedOrders*(R)
% The Float trait will be included in [8]

A.4 Integer

Integer: **trait**
  includes *TotalOrder*(*Int*)
  introduces
    $0, 1\colon\ \to Int$
    $succ, pred, -\_\_\colon Int \to Int$
    $\_\_ + \_\_, \_\_ - \_\_, \_\_ * \_\_\colon Int, Int \to Int$
  asserts
    Int **generated by** 0, *succ*, *pred*
    $\forall\ x, y\colon Int$
      $succ(pred(x)) == x$
      $pred(succ(x)) == x$
      $1 == succ(0)$
      $x + 0 == x$
      $x + succ(y) == succ(x + y)$
      $x + pred(y) == pred(x + y)$
      $-0 == 0$
      $-succ(x) == pred(-x)$
      $-pred(x) == succ(-x)$
      $x - y == x + (-y)$
      $x * 0 == 0$
      $x * succ(y) == x + (x * y)$
      $x * pred(y) == (-x) + (x * y)$
      $x < succ(y) == x \leq y$

A.5 Set

$Set(E, C)$: **trait**
  includes
    SetBasics,
    Integer,
    $DerivedOrders(C, \subseteq \textbf{ for } \leq, \supseteq \textbf{ for } \geq, \subset \textbf{ for } <, \supset \textbf{ for } >)$
  introduces
    $\_\_ \notin \_\_ : E, C \to Bool$
    $delete : E, C \to C$
    $\{\_\_\} : E \to C$
    $\_\_ \cup \_\_, \_\_ \cap \_\_, \_\_ - \_\_ : C, C \to C$

$size : C \rightarrow Int$

asserts

$\forall\ e, e_1, e_2 : E, s, s_1, s_2 : C$

$e \notin s == \neg(e \in s)$

$\{e\} == insert(e, \{\})$

$e_1 \in delete(e_2, s) == e_1 \neq e_2 \wedge e_1 \in s$

$e \in (s_1 \cup s_2) == e \in s_1 \vee e \in s_2$

$e \in (s_1 \cap s_2) == e \in s_1 \wedge e \in s_2$

$e \in (s_1 - s_2) == e \in s_1 \wedge e \notin s_2$

$size(\{\}) == 0$

$size(insert(e, s)) ==$ **if** $e \notin s$ **then** $size(s) + 1$ **else** $size(s)$

$s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

implies

$AbelianMonoid(\cup$ **for** $\circ, \{\}$ **for** $unit, C$ **for** $T)$,

$AC(\cap, C)$,

$JoinOp(\cup, \{\}$ **for** $empty)$,

$MemberOp(\{\}$ **for** $empty)$,

$PartialOrder(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq, \subset$ **for** $<, \supset$ **for** $>)$

C **generated by** $\{\}, \{\_\}, \cup$

$\forall\ e : E, s, s_1, s_2 : C$

$insert(e, s) \neq \{\}$

$insert(e, insert(e, s)) == insert(e, s)$

$s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

$size(s) \geq 0$

converts $\in, \notin, \{\_\}, delete, size, \cup, \cap, - : C, C \rightarrow C, \subseteq, \supseteq, \subset, \supset$

A.6 Stack

$Stack(E, C)$: **trait**

includes $Integer$

introduces

$empty : \rightarrow C$

$push : E, C \rightarrow C$

$top : C \rightarrow E$

$pop : C \rightarrow C$

$len : C \rightarrow Int$

$isEmpty : C \rightarrow Bool$

asserts

C **generated by** *empty*, *push*
$\forall\ e : E, stk : C$
$top(push(e, stk)) == e$
$pop(push(e, stk)) == stk$
$len(empty) == 0$
$len(push(e, stk)) == len(stk) + 1$
$isEmpty(stk) == stk = empty$
implies
OrderedContainer(*push* **for** *insert*, *top* **for** *head*, *pop* **for** *tail*)
C **partitioned by** *top*, *pop*, *len*
$\forall\ e : E, stk : C$
$len(stk) \geq 0$
$\neg isEmpty(push(e, stk))$
converts *top*, *pop*, *len*
exempting *top*(*empty*), *pop*(*empty*)

A.7 Array

Array(*V*, *VArray*): **trait**
introduces
$\_\_[\_\_] : VArray, Index \to VVar$
$\_\_$\pre, $\_\_$\post : $VArray \to VVec$
$\_\_[\_\_] : VVec, Index \to V$

A.8 Ref

Ref(*T*, *TRef*): **trait**
introduces
$\_\_$\pre, $\_\_$\post : $TRef \to T$
$narrow : RefAny \to TRef$
$widen : TRef \to RefAny$
$isTRef : RefAny \to Bool$
$Nil : \to TRef$
asserts
$\forall\ tr : TRef$
$isTRef(widen(tr))$
$narrow(widen(tr)) == tr$
% note: for any T1 not equal T, not(isTRef(widen(t1: T1Ref)))

A.9 Text

Text(*String*) : **trait**
introduces
fromString : *String* → *Text*
__&__ : *Text*, *Text* → *Text*
% This trait is incomplete.
% The full version will be included in [8]

A.10 Thread

ThreadTrait : **trait**
introduces
alertPending : *Th* → *Bool*

A.11 Mutex

Mutex : **trait**
includes (*Thread*)
introduces
none :→ *Th*
holder : *Mu* → *Th*

Appendix B

A parsing grammar

This grammar is presented in a format due to Bill McKeeman (of Digital's Technical Languages and Environments Group)[10]. It can be processed by a tool[1] into a variety of forms, such as a YACC grammar.

```
interface:
    INTERFACE ident ; traitUse imports intCons declarations END ident .
    INTERFACE ident ; traitUse         intCons declarations END ident .

traitUse:
     <* USING traitList *>

intCons:

     initially
     invariant
     initially invariant

imports:
     import
     imports import

import:
     FROM ident IMPORT idList ;
     IMPORT idList ;

declarations:
```

[1]This tool is freely available. Anyone interested in a copy should send me mail at kjones@src.dec.com.

```
     declaration
     declarations declaration

declaration:
     CONST constDeclarations
     TYPE typeDeclarations
     EXCEPTION exceptionDeclarations
     VAR variableDeclarations
     procedureDeclaration
     REVEAL typeDeclarations
     <* specVarDeclarations *>

constDeclarations:
     constDeclaration
     constDeclarations constDeclaration

constDeclaration:
     ident : type = constExpr ;
     ident = constExpr ;

idTypeDeclaration:
     : type

typeDeclarations:
     typeDeclaration
     typeDeclarations typeDeclaration

typeDeclaration:
     ident typeSpec subTypeRelation type ;
     ident subTypeRelation type ;

exceptionDeclarations:
     exceptionDeclaration
     exceptionDeclarations exceptionDeclaration

exceptionDeclaration:
     ident ;
     ident ( type ) ;

variableDeclarations:
     variableDeclaration
     variableDeclarations variableDeclaration

variableDeclaration:
```

```
     idList : type ;
     idList : type initialValue ;

specVarDeclarations:

     PRIVATE specVarDeclaration

specVarDeclaration:
     idList : sort varSpec
     specVarDeclaration ; idList : sort varSpec

varSpec:

     initially
     invariant
     initially invariant

initially:
     INITIALLY predicate

invariant:
     INVARIANT predicate

initialValue:
     := expr

subTypeRelation:
     =
     <:

typeSpec:
     <* BASED ON sortAndVar *>
     <* BASED ON sortAndVar invariant *>
     <* BASED ON sortAndVar initially *>
     <* BASED ON sortAndVar initially invariant *>

sortAndVar:
     sort
     ident : sort

procedureDeclaration:
     procedureHead ;
     procedureHead ; procedureSpec
```

```
procedureHead:
     PROCEDURE ident signature

signature:
     ( )
     ( ) resultType
     ( ) raisesList
     ( ) resultType raisesList
     ( formals )
     ( formals ) resultType
     ( formals ) raisesList
     ( formals ) resultType raisesList

formals:
     formal
     formals ; formal

formal:
     idList : type
     idList : type initialValue
     parameterType idList : type
     parameterType idList : type initialValue

parameterType:
     VALUE
     VAR
     READONLY

resultType:
     : type

raisesList:
     RAISES { }
     RAISES { exceptionIdList }

procedureSpec:
     <* globals specVarDeclarations letDeclarations prePred modifiesPred whenPred postPred *>

globals:

     globals RD idList : type ;
     globals WR idList : type ;

letDeclarations:
```

```
     LET letDecs IN

letDecs:
     ident BE expr
     letDecs , ident BE expr

modifiesPred:

     MODIFIES termList
     MODIFIES NOTHING
     MODIFIES ANY

prePred:

     REQUIRES predicate

whenPred:

     WHEN predicate

postPred:
     atomicPost
     compositePost

atomicPost:
     ensuresPost
     ensuresPost exceptPost
     ensuresPost unlessPost
     ensuresPost exceptPost unlessPost

compositePost:
     compositionDefinition actionsDeclarations

ensuresPost:
     ENSURES predicate

exceptPost:
     EXCEPT guardedExceptionPreds

unlessPost:
     UNLESS unguardedExceptionPreds

compositionDefinition:
```

```
     COMPOSITION OF actionsList END

actionsList:
     actionId
     actionsList ; actionId
     ( actionsList )

actionId:
     ident
     ident *

actionsDeclarations:
     actionDeclaration
     actionsDeclarations actionDeclaration

actionDeclaration:
     ACTION ident whenPred atomicPost

guardedExceptionPreds:
     predicate => predicate
     predicate => predicate | guardedExceptionPreds

unguardedExceptionPreds:
     predicate
     predicate | unguardedExceptionPreds

type:
     typeName
     typeName simpleObjectTypeList
     UNTRACED simpleObjectTypeList
     simpleObjectTypeList
     arrayType
     packedType
     enumType
     procedureType
     recordType
     refType
     setType
     subrangeType
     ( type )

simpleObjectTypeList:
     simpleObjectType
     simpleObjectTypeList simpleObjectType
```

```
arrayType:
     ARRAY OF type
     ARRAY [ typeIdList ] OF type

packedType:
     BITS constExpr FOR type

enumType:
     { }
     { idList }

simpleObjectType:
     OBJECT methodDeclarations END
     OBJECT fields methodDeclarations END
     brand OBJECT methodDeclarations END
     brand OBJECT fields methodDeclarations END

methodDeclarations:

     METHODS methodSpecs

procedureType:
     PROCEDURE signature

recordType:
     RECORD fields END

refType:
     UNTRACED REF type
     REF type
     UNTRACED brand REF type
     brand REF type

setType:
     SET OF type

subrangeType:
     [ constExpr .. constExpr ]

brand:
     BRANDED
     BRANDED brandName
```

```
fields:
     field
     fields field

field:
     idList idTypeDeclaration initialValue ;

methodSpecs:
     method
     methodSpecs method

method:
     explicitMethod
     strengthenMethodSpec

explicitMethod:
     ident signature defaultProc ; procedureSpec

defaultProc:

     := procedureId

strengthenMethodSpec:
     <* STRENGTHEN ident predicate *>

constExpr:
     expr

expr:
     term

predicate:
     term

termList:
     term
     termList , term

term:
     IF term THEN term ELSE term
     quantifiedTerm
     logicalTerm

quantifiedTerm:
```

```
     quantifier boundVarDeclarationList ( term )

quantifier:
     FORALL
     EXISTS

boundVarDeclarationList:
     idList : type
     idList : STATE

logicalTerm:
     equalityTerm
     logicalTerm logicalSym equalityTerm

equalityTerm:
     simpleOpTerm
     simpleOpTerm = simpleOpTerm
     simpleOpTerm eqSym simpleOpTerm

simpleOpTerm:
     prefixOpTerm
     secondary postfixOps
     secondary infixOpTerm

postfixOps:
     simpleOp
     postfixOps simpleOp

infixOpTerm:
     simpleOp secondary
     infixOpTerm simpleOp secondary

prefixOpTerm:
     secondary
     simpleOp prefixOpTerm

primary:
     ( term )
     simpleId
     simpleId ( termList )
     primary selectSym simpleId
     primary : sort
     literal
```

```
secondary:
     primary
     bracketed
     bracketed primary
     primary bracketed
     primary bracketed primary

bracketed:
     matched : sort
     matched

matched:
     beginParen args endParen
     beginParen endParen

beginParen:
     [
     openSym

endParen:
     ]
     closeSym

args:
     term
     args sepSym term
     args , term

simpleId:
     ident

typeId:
     typeName

typeName:
     qualifiedId
     ROOT
     UNTRACED ROOT

brandName:
     textLiteral

exceptionIdList:
     exceptionId
```

```
     exceptionId , exceptionIdList

exceptionId:
     qualifiedId

procedureId:
     qualifiedId

idList:
     ident
     idList , ident

typeIdList:
     typeId
     typeIdList , typeId

trait:
     traitId
     traitId ( renaming )

renaming:
     replaceList
     nameList
     nameList , replaceList

nameList:
     name
     nameList , name

replaceList:
     replace
     replaceList , replace

replace:
     name FOR name
     name FOR name opSignature

name:
     qualifiedId

opSignature:
     : domain mapSym range

domain:
```

```
     sortList

sortList:
     sort
     sortList , sort

range:
     sort

traitId:
     ident

traitList:
     trait
     traitList , trait

sort:
     ident

qualifiedId:
     ident
     ident . ident

ident:
     IDENTIFIER

literal:
     TEXTLITERAL
     STRINGLITERAL
     NUMERICLITERAL
     BOOLEANLITERAL
```

Bibliography

[1] A.D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin, "Thread synchronization: a formal specification", pp. 119–129 in [11].

[2] A. D. Birrell., "An introduction to programming with Threads", pp. 88–118 in [11].

[3] Mark R. Brown and Greg Nelson, "IO Streams: Abstract Types, Real Programs", SRC Report 53, 1989.

[4] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson, "Modula-3 Report (revised)", SRC Report 52, 1989.

[5] S.J. Garland and J.V. Guttag, 'An Overview of LP, The Larch Prover,' Proc. 3rd Intl. Conf. Rewriting Techniques and Applications, LNCS 355, 1989, pp. 137-151.

[6] J.V. Guttag, J.J. Horning, and A. Modet, "Report on the Larch Shared Language, Version 2.3", SRC Report 58, 1990.

[7] J.V. Guttag and J.J. Horning (eds.), "The Larch Book", in preparation.

[8] J.J. Horning, "A Larch Shared Language Handbook", in [7].

[9] Leslie Lamport, "A Temporal Logic of Actions", SRC Report 57, 1990.

[10] W.M. McKeeman, private communication, 1990.

[11] Greg Nelson (ed.), "Systems Programming with Modula-3", Prentice Hall, 1991.

[12] Jeannette M. Wing, "A Two-Tiered Approach To Specifying Programs", MIT/LCS/TR-299, 1983.

[13] Jeannette M. Wing, "A Specifier's Introduction to Formal Methods", Computer, Volume 23, Number 9, September 1990.