

Chapter 2

The CARE language and development methodology

2.1 Introduction

In this thesis a unified approach to adaptation and retrieval of formally specified reusable components is presented. As a vehicle for further exploring these ideas the CARE (which stands for **Computer Assisted Refinement Engineering**) language and toolset will be used. The development of the CARE method and tools was done as part of a collaborative project between the Software Verification Research Centre (SVRC) and Teletronics Pacing Systems (see [3, 61, 40, 60, 59] for more details on CARE). The author of this thesis was involved in the collaboration as the chief or co-designer and implementor of many of the tools, as well as being involved in the specification and design of much of the language.

The CARE language is relatively simple; however it has analogies to many of the features of other formal languages used for software development and assurance, making the CARE language a good choice for exploring the ideas presented in this thesis.

For the purpose of this thesis, the CARE language is partitioned into three separate levels of constructs - expressions, units and modules. The main features, as found in many formal languages, of each of these separate levels of constructs are

listed below.

expressions: include variables, sets, functions, relations, quantifiers and (higher-order) parameters;

units: include operations, data types, definitions and theorems. Units can be parameterised (over both formal and textual parameters). Units have separate specification and implementation parts, meaning that implementation details can be hidden. Units can be implemented either using target code structures, or by calling other units (in particular data and algorithm refinements are supported).

modules: collections of related units can be encapsulated in a single *template* which implements an algorithm, data refinement, theories, or provides access to “primitive” components.

An abstract syntax for the CARE language has been formally specified [37], including a formal definition of the semantics. The abstract syntax has been used as the basis for specifying a number of CARE tools (see [39] for an example). The result of this is that the tools could be formally specified, making prototyping (and indeed full implementation) of the tools more straightforward. In a similar vein, the abstract syntax will be used in this thesis to define the tools for supporting reuse in CARE. While the methods and tools developed for reuse in this thesis are closely dependent on the abstract syntax, it should be noted that the abstract syntax for CARE contains many of the basic features of other formal languages, and so these methods and tools can easily be generalised beyond the scope of CARE, without getting bogged down with matters of concrete syntax.

Another reason for choosing CARE as the vehicle for exploring the ideas of this thesis is that it comes with a well-defined development methodology and a set of tools for supporting the methodology (and the author is very familiar with both). Indeed the thesis was borne out of the belief that for CARE to be practicable, support for using the library components needed to be significantly improved. The existence

of such a toolset means that the ideas presented in this thesis could be tested and developed in a concrete sense in the form of additions and enhancements to the existing toolset.

It should be noted that the design of the CARE language and development methodology is not part of the thesis. The design was done as part of a previous project; with the exception of a few minor changes the language and methodology will be used “as is” in this thesis. CARE itself is not the subject of this thesis, it is just the language used to illustrate the ideas in this thesis.

The CARE language is described in Sections 2.2-2.4 of this chapter. The description includes formal specifications of an abstract syntax for the language (using the Z specification language) for those constructs used later in thesis. A simple, open style of specification will be used for the abstract syntax. Constraints (such as function arities, preconditions and expression-level type checking) will not be modelled in detail here. For a full formal specification of the CARE language (including a formal definition of the semantics) the reader is referred to a technical report by Hemer and Lindsay [37]. The concrete syntax for the CARE language will be explained and used, however it will not be formally specified.

The CARE methodology is briefly summarised in Section 2.5, giving the context in which the library search tools will be developed. The individual steps required to develop a verified program in CARE are described, starting with a formal specification of the program together with some domain specific theory. The tools which support the user in developing a verified program in CARE are summarised in Section 2.6. Finally in Section 2.7, a description is given of each of the reusable library components used in this thesis.

2.2 Expressions

The mathematical language used in CARE is a form of higher-order logic, with parameters ranging over functions, relations and sets. The mathematical language is extensible, with the user being able to declare new functions, relations and sets,

as well as introducing assertions about the constructs. There is some predefined theory, based on the extensions to the Z Mathematical Toolkit [87]; more details on this predefined theory, in particular the extensions used in this thesis, are given in Appendix A. This appendix also gives the ASCII counterparts (used in illustrating the prototype tools) to any of the special mathematical symbols used in this thesis.

2.2.1 Sets

Set-valued expressions, used to represent a collection of values, are split into three separate categories in CARE - *given* sets, *constructed* sets and *parametric* sets. Given sets are collections of values which are not further defined, they are represented by a set name. Examples of given sets include the set of natural numbers and the set of characters. Constructed sets are built from one or more other (simpler) sets using a *set constructor*, they are represented by a set constructor and one or more other sets. Parametric sets are sets whose values have not yet been fixed, they are represented by a set parameter.

Given sets and parametric sets are represented by identifiers from *SetName* and *SetParam* respectively. The convention in CARE is to represent these sets as ASCII strings beginning with an uppercase character (e.g. *X*, *CHAR*, *Y* etc.), in some cases special symbols will also be used (i.e. *N* is used to represent the set of natural numbers), however the use of such symbols is restricted to commonly used symbols.

$$[SetName, SetParam]$$

Set-valued expressions are modelled here as a free type consisting of *given* sets, *constructed* sets and *parametric* sets.

$$Set ::= \begin{array}{l} \text{given}\langle\langle SetName \rangle\rangle \\ \quad | \quad \text{constructor}\langle\langle SetConstructor \times \text{seq}_1 Set \rangle\rangle \\ \quad | \quad \text{paramset}\langle\langle SetParam \rangle\rangle \end{array}$$

Constructed sets are built from one or more other sets using a type constructor; examples of type constructors include sequences, finite power sets and power sets.

$$SetConstructor == \{\text{'seq'}, \text{'F'}, \text{'P'}, \dots\}$$

Example 2.1 The set ‘ $\text{constructor}(\mathbb{F}, \langle \text{given } \mathbb{N} \rangle)$ ’ is represented in concrete notation as $\mathbb{F}\mathbb{N}$ (the finite set of natural numbers).

2.2.2 Terms

The sets Var , $FunctionName$ and $FunctionParam$ represent the name sets for variables, function names and function parameters respectively. Variables are represented by lower case identifiers, often consisting of a single character. Function names are either represented by lower case identifiers or by a special symbols, such as those used in the Z Mathematical toolkit (which are also given an ASCII counterpart). Function parameters are represented by lower case identifiers.

$$[Var, FunctionName, FunctionParam]$$

Terms are those expressions in CARE which have a single (non-boolean) value. A term can be either a *placeholder*, a *variable*, a (*non-parametric*) *function* application or a *parametric function* application. Placeholders are used to define instantiations of function and relation parameters, they are represented abstractly as a natural number. For example given an instantiation of a parameter F to the term $\text{fnapplic}(g, \langle \text{ph}(2), \text{ph}(1) \rangle)$, means that an application of F (which must have at least two arguments) is replaced by the function g applied to the second and first arguments of F . In concrete syntax, $F(a, b) \rightsquigarrow g(\langle b, a \rangle)$ where a and b are symbolic placeholders.

Functions map a number of values (the arguments) to a single value; an application of a function is represented by a function name and a sequence of terms. Terms can be parameterised over functions by including a parametric function application; represented abstractly by a function parameter and a sequence of terms.

$$\begin{array}{l} Term ::= \text{ph}\langle\langle \mathbb{N}_1 \rangle\rangle \\ \quad | \text{var}\langle\langle Var \rangle\rangle \\ \quad | \text{fnApplic}\langle\langle FunctionName \times \text{seq } Term \rangle\rangle \\ \quad | \text{termparam}\langle\langle FunctionParam \times \text{seq } Term \rangle\rangle \end{array}$$

Note that symbolic placeholders (e.g. \mathbf{x}, \mathbf{y}) are used in examples, but for convenience of specification they are modelled here as non-zero natural numbers.

Example 2.2 The abstract term $\text{fnApplic}(f, \langle \text{var } x, \text{fnApplic}(g, \langle \rangle) \rangle)$ has the concrete representation $f(x, g)$.

Each function has an associated *signature* which gives the mathematical types of its domain and the mathematical type of its codomain. Note that constants are treated as functions with null (empty) domains.

FunctionSig $\text{domain} : \text{seq } \text{Set}$ $\text{codomain} : \text{Set}$
--

2.2.3 Formulae

The types *RelationName* and *RelationParam* represent the name sets of relations and relation parameters. Relations are represented as either identifiers beginning with an upper case character (e.g. *Disjoint*), or special symbols such as those used in Z (e.g. \in , \subseteq). Relation parameters are represented by identifiers starting with an upper case character (e.g. *R*).

$[\text{RelationName}, \text{RelationParam}]$

CARE formulae are modelled here as a free type consisting of the ‘true’ formula, negation of a formula, connection of two formulae, formula quantification, (non-parametric) relation applications and parametric relation applications .

$$\begin{array}{l}
 \text{Fmla} ::= \text{true} \\
 \quad | \text{not}\langle\langle \text{Fmla} \rangle\rangle \\
 \quad | \text{binaryConn}\langle\langle \text{BinConn} \times \text{Fmla} \times \text{Fmla} \rangle\rangle \\
 \quad | \text{quant}\langle\langle \text{Quant} \times \text{MathVarDeclar} \times \text{Fmla} \rangle\rangle \\
 \quad | \text{relation}\langle\langle \text{RelationName} \times \text{seq } \text{Term} \rangle\rangle \\
 \quad | \text{paramrelation}\langle\langle \text{RelationParam} \times \text{seq } \text{Term} \rangle\rangle
 \end{array}$$

Two formulae can be connected to give a single formula using one of the logical binary connectives - conjunction, disjunction, equivalence and implication. Quantifiers that are modelled include universal and existential quantifiers.

$$\begin{array}{l}
 \text{BinConn} == \{ \wedge, \vee, \Leftrightarrow, \Rightarrow, \dots \} \\
 \text{Quant} == \{ \forall, \exists, \dots \}
 \end{array}$$

Note that binary connectives will normally be written in infix notation. Variables are declared within a *mathematical variable declaration list*, where each variable is associated with a set of values that it may take.

$$\mathit{MathVarDeclar}s ::= \text{seq}_1(\mathit{Var} \times \mathit{Set})$$

Example 2.3 The formula $x = y \wedge Q(x, y)$ is represented as

$$\text{binaryConn}(\wedge, \text{relation}(=, \langle \text{var } x, \text{var } y \rangle), \text{relation}(Q, \langle \text{var } x, \text{var } y \rangle))$$

The signature for a relation gives the mathematical types of its domain.

$\begin{array}{l} \mathit{RelationSig} \\ \text{domain} : \text{seq } \mathit{Set} \end{array}$

2.3 Formally specified units

The next level of components in CARE are *units*, which include types, fragments, operator declarations and assertions. Types correspond to data structures; fragments correspond roughly to functions and procedures in a procedural programming language; and assertions correspond to definitions, lemmas and proof obligations (explained below).

$$\begin{array}{l} \mathit{Unit} ::= \text{fragment}\langle\langle \mathit{Fragment} \rangle\rangle \\ \quad | \text{type}\langle\langle \mathit{Type} \rangle\rangle \\ \quad | \text{opdecl}\langle\langle \mathit{OperationDecl} \rangle\rangle \\ \quad | \text{assertion}\langle\langle \mathit{Assertion} \rangle\rangle \end{array}$$

Each unit is uniquely identified by a name from the name sets *FragmentName*, *TypeName*, *OpName* and *AssertionName* defined later.

$$\mathit{UnitName} ::= \mathit{FragmentName} \cup \mathit{TypeName} \cup \mathit{OpName} \cup \mathit{AssertionName}$$

Each unit has its own formal specification, which may include constraints on how it can be used. Units are classified as either *primitive* or *higher-level*. In essence, primitive units are those whose proof of correctness is outside the scope of CARE, while higher-level units have associated proof obligations. More specifically:

Primitive units are supplied as part of the library, and are not typically written by the ordinary user. Primitive types and fragments are implemented directly in the target programming language and provide access to target language data structures and basic functionality. (B uses a similar approach [55].) The CARE specification of such a component describes the mathematical properties that may be assumed for the component. Verification of such properties is outside the scope of CARE, and would depend on having a mathematical model of the semantics of the target language and its compiler: a primitive type's specification describes the set of mathematical values corresponding to the associated data structure; a primitive fragment's specification describes the associated target code's functionality.

Higher-level units are constructed from other units within the CARE framework. Higher-level types and fragments express data refinements and algorithm designs respectively, and are implemented in the CARE language. CARE tools generate proof obligations which show that the unit's implementation is correct with respect to its specification. The different kinds of proof obligations will be explained below under 'verification'.

Note that the focus of this section is on unit specifications rather than implementations since they are more important for this thesis. In particular the implementations of fragments and types will be described and illustrated with examples, but will not be formally modelled in this thesis. More details on the abstract syntax for unit implementations in CARE are given in an earlier technical report [37].

During development a CARE program may contain units which have specifications but which do not yet have implementations. A *complete* program is one in which all units are implemented.

2.3.1 Operator declarations and assertions

New mathematical constructs are introduced in CARE by giving an operator declaration which specifies the operator name and its signature:

$$\begin{array}{l}
 \text{OperatorDecl} ::= \text{functdec}\langle\langle \text{FunctionName} \times \text{FunctionSig} \rangle\rangle \\
 \quad | \text{reldec}\langle\langle \text{RelationName} \times \text{RelationSig} \rangle\rangle \\
 \quad | \text{setdec}\langle\langle \text{SetName} \rangle\rangle \\
 \quad | \text{functparamdec}\langle\langle \text{FunctionParam} \times \text{FunctionSig} \rangle\rangle \\
 \quad | \text{relparamdec}\langle\langle \text{RelationParam} \times \text{RelationSig} \rangle\rangle \\
 \quad | \text{setparamdec}\langle\langle \text{SetParam} \rangle\rangle
 \end{array}$$

Each operator declaration has an associated name, usually just the name of the operator being defined.

$$\begin{array}{l}
 \text{OpName} == \text{FunctionName} \cup \text{RelationName} \cup \text{SetName} \cup \\
 \quad \text{FunctionParam} \cup \text{RelationParam} \cup \text{SetParam}
 \end{array}$$

Theorems, axioms, lemmas, applicability conditions and proof obligations are all represented in the CARE abstract syntax as *assertions*. Each assertion consists of a name and a statement, the latter being represented by a formula.

$$[\text{AssertionName}]$$

$ \begin{array}{l} \text{Assertion} \\ \text{name} : \text{AssertionName} \\ \text{statement} : \text{Fmla} \end{array} $

Operator declarations and axioms/definitions are expressed in the CARE concrete notation within a *theory*. A theory is a collection of operator declarations and assertion units (similar to an axiomatic definition in Z). Axioms within a theory are not given a name explicitly, however a name can be constructed by combining a string formed from the theory name with a number representing the position of the axiom in the list of axioms contained in the theory to give a unique identifier. Other kinds of assertion units such as lemmas and proof obligations are given separately.

Fig. 2.1 gives a theory for the *append* function, which consists of declaration of the *append* operator, together with a definition of *append* in terms of list concatenation. Next a lemma which relates *append* with the operators *head* and *tail* is given. Finally a lemma is given which relates the *append* and *ran* functions.

As with the other kinds of units, assertions can be broken into primitive and higher-level units. Primitive assertion units correspond to definitions and axioms; higher-level units correspond to lemmas, proof obligations and applicability conditions which are “implemented” by proofs.

Theory definition of function *append*.

$append : Elem \times seq\ Elem \rightarrow seq\ Elem;$

$\forall h : Elem; t : seq\ Elem \bullet append(h, t) = \langle h \rangle \frown t,$

Lemma *append_head_tail*.

$\forall s : seq\ Elem \bullet \#s \neq 0 \Rightarrow append(head(s), tail(s)) = s.$

Lemma *ran_of_append*.

$\forall e : X; s : seq\ X \bullet ran(append(e, s)) = (ran\ s) \cup \{e\}$

Figure 2.1: Example assertions in CARE

2.3.2 Types

A CARE type consists of a name, a specification and an implementation, explained separately below.

[*TypeName*]

Type

name : *TypeName*

spec : *Set*

body : *TypeBody*

The specification of a CARE type is an expression denoting the set of mathematical values that objects of the type can take. The example given in Fig. 2.2 contains specifications of CARE types. The first line declares the CARE type `Index`, which is modelled mathematically as the set of natural numbers. The second line declares the type `Element` which corresponds to some given type E . The third line declares the type `List` which corresponds to a sequence of natural numbers.

Type `Index` has specification: \mathbb{N} .

Type `Element` has specification: E .

Type `List` has specification: $seq\ \mathbb{N}$.

Figure 2.2: Example CARE type specifications

2.3.2.1 Variable declarations

Before describing type implementations, an expression for introducing *variable declarations*, in which new CARE variables are introduced, must be defined. *VarDeclars* associates a unique CARE type with each variable. A variable's type describes the set of values that a variable may take. Note that there can be no duplication of variable names within a variable declaration list.

$$\begin{aligned} \text{VarDeclars} ::= & \{s : \text{seq}(\text{Var} \times \text{TypeName}) \mid \\ & \forall i, j : \mathbb{N} \bullet 1 \leq i < j \leq \#s \Rightarrow \text{first } s(i) \neq \text{first } s(j)\} \end{aligned}$$

2.3.2.2 Type implementations

A *TypeBody* gives a description of how a type is implemented. It is either a primitive type body (implemented directly in target code), or a refined type body, implemented in terms of other types.

$$\begin{aligned} \text{TypeBody} ::= & \text{primitive}\langle\langle \text{PrimitiveTypeBody} \rangle\rangle \\ & \mid \text{refined}\langle\langle \text{RefinedTypeBody} \rangle\rangle \end{aligned}$$

Primitive types are implemented by some target language data structure. Details on the implementation of primitive types is outside of the scope of this thesis. A higher-level type (the *refined type*) is implemented in terms of one or more other types (the corresponding *concrete types*) by data refinement [43]. The specification of a refined type describes the relationship between values of the refined type and their concrete representations (the *refinement relation*), an optional condition restricting the values that the refined type may take (the *constraint*), and an optional condition restricting the values the concrete types may take (the *invariant*).

RefinedTypeBody _____

<i>refvalue</i> : <i>Var</i> <i>constraint</i> : <i>Fmla</i> <i>outvars</i> : <i>VarDeclars</i> <i>refrel</i> : <i>Fmla</i> <i>invariant</i> : <i>Fmla</i>
--

Note that the refinement relation can be non-functional, in line with Back's [6] generalisation. This means that each concrete value can be associated with more

than one abstract value. An example showing sets refined as non-repeating lists is given in Fig. 2.3. Here the refinement relation states that the set is equal to the range of the underlying list. An invariant on the list is given stating that the list cannot contain any repetitions.

Type `Set` has specification: $\mathbb{F} X$
 refinement:
 value `s` is refined by `l:List` with invariant $IsNonRep(l)$
 with refinement relation $s = \text{ran } l$

Figure 2.3: Refined type

2.3.2.3 Type verification

For refined types, there are proof obligations to check that the refinement relation defines a function whose domain is given by the invariant and whose range is given by the constraint. CARE tools generate the assertion units which embody these proof obligations (see [39] for more details).

2.3.3 Fragments

A fragment consists of a name, specification and body. The name uniquely identifies the fragment, the specification gives a mathematical description of the fragment (see Section 2.3.3.1), while the body gives the implementation of the fragment. *FragmentName* represents the name set for fragments. By convention fragment names are usually a string beginning with a lower case character (e.g. `reverse`, `bubble_sort`). In some cases special symbols, corresponding to commonly used mathematical operators are also used (e.g. `+`, `*`).

$[FragmentName]$

There are two kinds of fragments: *simple* and *branching*.

$$Fragment ::= \text{simple}\langle\langle SimpleFragment \rangle\rangle \\ \quad \quad \quad | \text{branching}\langle\langle BranchingFragment \rangle\rangle$$

Simple fragments correspond roughly to functions in a procedural programming language; they take inputs and return outputs.

<i>SimpleFragment</i> <i>name</i> : <i>FragmentName</i> <i>spec</i> : <i>SimpleFragmentSpec</i> <i>body</i> : <i>FragBody</i>
--

The specification of branching fragments differs from simple fragments in that the number and kind of outputs returned can differ depending on the inputs; thus the specification has a number of *branches*. The implementation of branching fragments differ from simple fragments since each result returned in the implementation must have a pointer back to one of the branches in the specification. These points are described in more detail below.

<i>BranchingFragment</i> <i>name</i> : <i>FragmentName</i> <i>header</i> : <i>BFragmentSpec</i> <i>body</i> : <i>FragBody</i>
--

Remark: The abstract syntax for branching fragments given below (in particular the specification of branching fragments) is general enough to also model the simple fragment case. However modelling simple fragments separately results in a cleaner and simpler abstract syntax, which means that the illustration of ideas later in the the thesis is much easier.

2.3.3.1 Fragment specifications

The mathematical description of a fragment is contained within a specification. The specification of a simple fragment consists of a name, a declaration of its input variables, an optional *precondition*, a declaration of its output variables, and the required input/output relationship (or *postcondition*). The number and type of inputs and outputs is fixed. The example given in Fig. 2.4 shows specifications for the simple fragments `nil`, `cons`, `car` and `cdr` for manipulating lists, in concrete syntax, using LISP-like naming conventions.

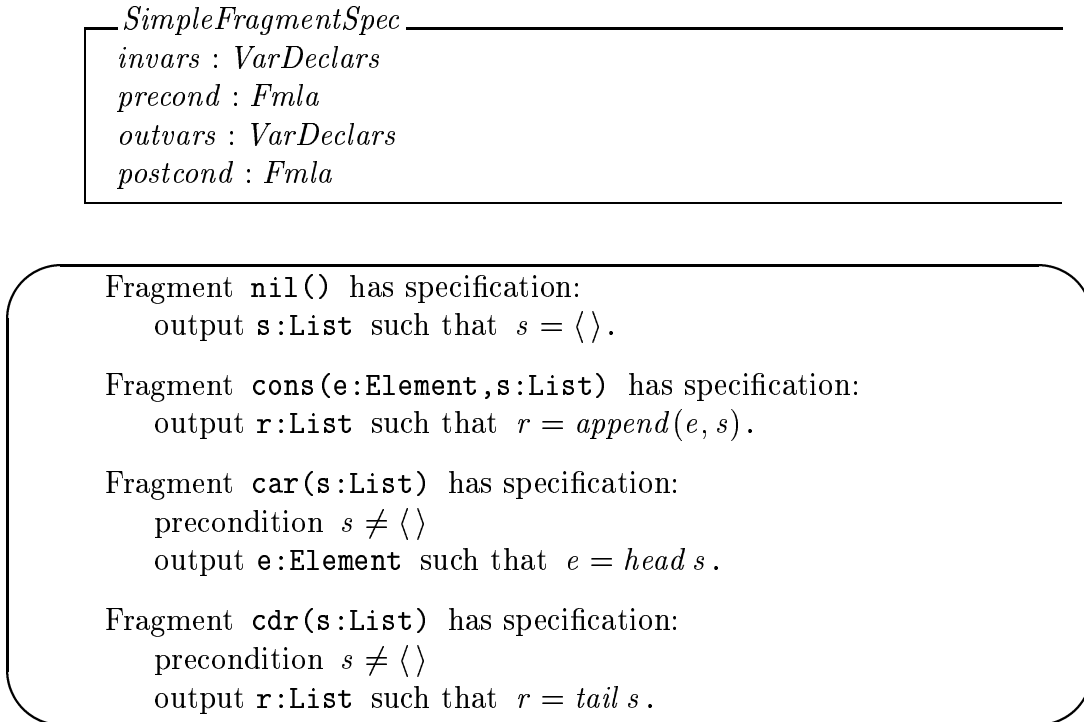
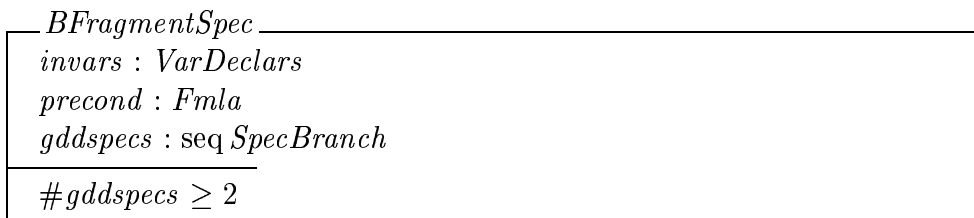


Figure 2.4: Simple fragment specifications

The specification of a branching fragment consists of its name, input variable declarations, an optional precondition and a sequence of guarded branches. Each branch contains a *test*, a description of the outputs and their types, an optional postcondition, and a *report*, which identifies the branch. Note that a branching fragment must contain at least two branches in its specification. (The test in the last branch is `true` by default.) In Fig. 2.5 are examples of specifications of branching fragments `search` and `decompose`. Note that the number and type of outputs on each branch is fixed but may differ from branch to branch.



For example, the `search(s, e)` fragment has two cases: when `e` occurs in `s`, it reports `found` and returns an index `i` at which `e` can be found; otherwise it simply reports `notfound` with no outputs. The *guard* of a branch is its test conjoined with

the negations of the tests of the preceding branches. For example the guard of the nonempty branch of `decompose(s)` is $\neg (\#s = 0)$.

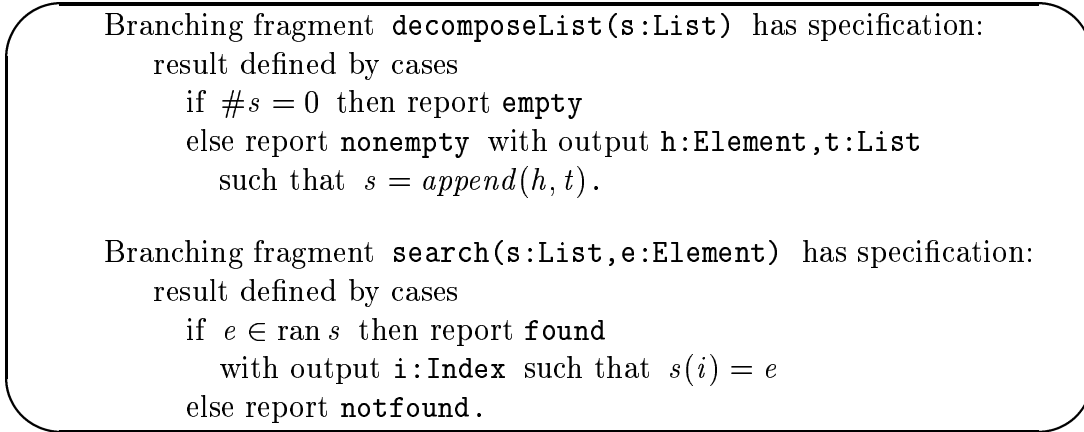


Figure 2.5: Branching fragment specifications

Note that fragment specifications may be under-determined, in the sense that more than one output may satisfy the postcondition for any given input: e.g. `i` in `search(s, e)`. In practice however, the postcondition is often an equation defining the output variables directly as a function of the input variables.

Remark: A non-standard feature of the CARE language is that branching fragments can return different numbers and kinds of outputs on different branches.

Specification branches A branching fragment has at least two specification branches, valid for different input values. To determine which specification branch is a valid description of the I/O relationship for a particular input value, a guard is associated with each specification part in the fragment's header. The guard is a formula giving the set of input values for which the associated specification branch is valid. To determine which branch is valid, the guards are evaluated sequentially, until one is found which the inputs satisfy. The last branch (referred to as the *else* branch), which is not given a guard explicitly, captures all inputs failing the previous guards. The guard for the else branch is equal to the negation of all previous guards. Guards are evaluated sequentially to avoid having to check that they are non-overlapping, which would complicate the methodology, and result in extra proof

obligations.

A *SpecBranch* consists of a guard, a report, a list of output variables and a post-condition. The report is represented by an identifier of some sort (not defined further here):

[*Report*]

<i>SpecBranch</i> <i>guard</i> : <i>Fmla</i> <i>report</i> : <i>Report</i> <i>outvars</i> : <i>VarDeclar</i> <i>postcond</i> : <i>Fmla</i>
--

Note that the prototype tools use numeric reports (e.g. 1, 2, etc.) while the last branch has the report “e” (indicating the *else* branch).

2.3.3.2 Fragment implementations

For completeness, fragment implementations will be explained in this section, however they will only be modelled as a primitive type here.

[*FragBody*]

The implementation of primitive fragments and higher-level fragments differ. These two cases are described separately below.

Primitive fragments are implemented by giving code segments in the target language. The exact nature of the code segments in primitive fragments is dependent on the target language, as well as the code synthesis tools. The current prototype in CARE uses low level C as the target language. The details of this target language is outside of the scope of this thesis, and since the target code would require considerable explanation it has been elided in the examples below.

Higher-level fragments are implemented in terms of calls to other fragments. The CARE implementation language supports the following simple design algorithm constructs: assignment of values to local variables, fragment calls, sequencing, branching of control, and data refinement transformations.

In effect, the body of a higher-level fragment is tree-structured. Non-branching nodes of the tree correspond to bindings to local variables of the values returned by simple fragment calls or variables. Branching nodes correspond to calls to branching fragments, labelled by the corresponding reports; where branches return values, these values are bound to local variables. Local variables are newly created at the point of assignment, meaning that they are similar to bound variables in quantified expressions. In particular local variables can be renamed without changing the meaning of the program, provided the renaming does not create clashes with existing variable names. (The scope of the local variables is that part of the implementation tree below the binding part, down to the leaves). The leaves of the tree define the fragment's output values, leaves inside the implementation of a branching fragment also contain a report. An *abort* statement is also provided, for use in branches which will never be executed. An example of higher-level fragment implementations is given in Fig. 2.6.

Recursive calls and mutual recursion are allowed, provided the recursion eventually terminates. To establish termination, the CARE user supplies a well-founded variant function (or *variant* for short) whose value decreases on recursive calls and is bounded below.

```

Fragment reverse(s:List) has specification:
  output r:List such that  $r = rev(s)$ 
implementation:
  return revAcc(s,nil).

Fragment revAcc(u:List,v:List) has specification:
  output w:List such that  $w = rev(u) \hat{\ } v$ 
implementation:
  cases decomposeList(u) of:
    empty:    return v
    nonempty: assign output to h:Element,t:List;
              return revAcc(t,cons(h,v)).

variant #u.

```

Figure 2.6: Part of a CARE program for reversing a list.

The bodies of higher-level branching fragments are similar in form to those of simple fragments, except that a report is produced at non-aborting leaves, together

with output values, if appropriate (see Fig. 2.7).

```

Branching fragment decomposeList(s:List) has specification:
result defined by cases
  if #s = 0 then report empty
  else report nonempty with output h:Element,t:List
    such that s = append(h,t)
implementation:
cases null(s) of:
  yes: report empty
  no: report nonempty and return car(s),cdr(s)

```

Figure 2.7: A fragment for decomposing lists.

2.3.3.3 Fragment verification

The purpose of fragment verification is to check that a fragment's implementation satisfies its specification. To a large extent, fragments are verified individually, which leads to an incremental style of working. This section outlines an informal semantics for fragments and describes how to reason about their correctness.

Verification of a fragment set involves establishing a number of *proof obligations*, which fall into four categories:

Partial correctness: The result returned at each (non-aborting) leaf of an implementation tree satisfies the appropriate postcondition.

Termination: For recursively-defined fragments, the variant is strictly decreasing on recursive calls. Since the variant is bounded below by zero, it cannot decrease indefinitely, so the recursion must eventually terminate.

Well-formedness: For each fragment call, the fragment's precondition (if any) is satisfied.

Non-execution: Execution cannot reach an 'abort' leaf (at least, not for input values which satisfy the fragment's precondition).

If all of the proof obligations can be shown to be logical consequences of the theory of the problem domain, the fragment set is guaranteed to be correct, in the sense that

execution of a fragment on input values which satisfy its precondition will terminate and return a result which satisfies the fragment's specified postcondition.

CARE tools generate the proof obligations by considering the different possible execution paths through the fragment (or through the fragment set, for the termination proof obligation when mutual recursion is present). For each path, the intermediate results returned by fragment calls are assumed to satisfy the appropriate postcondition. The interested reader is referred to [39] for a more detailed treatment of proof obligation generation and its justification.

2.4 Modular components

This section contains a description of CARE *templates* which are module-like structures. Templates are parameterised collections of units (fragments, types, assertions etc.), which collectively implement an algorithm, data refinement, or theory, or provide access to primitive components. Templates are typically proven off-line by a proof expert; as part of the proof process, applicability conditions on the parameters are generated which provide sufficient conditions to guarantee a template's correctness.

An example - the *accumulator* template which implements the commonly used list accumulation strategy - is used throughout this section to illustrate the ideas.

The *accumulator* template given in Fig 2.8 takes a list and returns some value, defined by a list processing fragment `processList` (represented mathematically by the function f). The algorithm is implemented by successively applying a so-called "step function" `processElem` (represented by the mathematical function hd), to each element in the list step-by-step, progressing from the first through to the last element, and accumulating an intermediate value at each stage. The accumulator is started with a given value `base` (represented by the function $base$).

The accumulator is parameterised over the functions f , hd and $base$, together with the type of the individual list elements E and the type of the accumulated values Acc . The user needs to instantiate these parameters to suitable values to

solve their particular problem. They also need to instantiate the logical function dh , which acts as a dual to the function hd .

Applicability conditions, representing the minimal conditions under which correctness of the template is preserved, are also given. These conditions must be shown to hold upon instantiation of the template. The first two conditions state that the function f must be definable in terms of $base$ and hd . The other two conditions state the duality conditions between hd and its logical counterpart. These conditions result from the process of proving the correctness of the template, which is done once off-line by a proof expert.

Finally the function $fold$ is defined, which models the process of applying the step-wise function hd recursively to the elements of the list from left to right, at each stage updating the accumulated value. Upon instantiation of the template, the definition of $fold$ is added to the user's program.

Example 2.4 To illustrate how the template would typically be used, let us consider the development of a program for summing a list of numbers specified as follows:

Type `Nat` has specification: \mathbb{N}

Type `NatList` has specification: $\text{seq } \mathbb{N}$

Fragment `sum(s:NatList)` has

specification: output `n:Nat` such that $n = \text{sum}(s)$.

where

Theory definition of sum .

$\text{sum} : \text{seq } \mathbb{Z} \rightarrow \mathbb{Z};$

$\text{sum}\langle \rangle = 0,$

$\forall x : \mathbb{Z} \bullet \text{sum}\langle x \rangle = x,$

$\forall s, t : \text{seq } \mathbb{Z} \bullet \text{sum}(s \hat{\ } t) = \text{sum}(s) + \text{sum}(t).$

The user would match these above components with the `Elem`, `List` and `processList` components in the template by supplying the following partial instantiation:

<u>Template: Accumulator</u>
<u>Formal parameters:</u>
$E, Acc, f : \text{seq } E \rightarrow Acc, hd : Acc \times E \rightarrow Acc, dh : E \times Acc \rightarrow Acc, base : E.$
<u>Applicability conditions:</u>
$f(\langle \rangle) = base,$ $\forall h : E, t : \text{seq } E \bullet f(\text{append}(h, t)) = dh(h, f(t)),$ $\forall x : E \bullet dh(x, base) = hd(base, x),$ $\forall x, y : E; a : Acc \bullet dh(x, hd(a, y)) = hd(dh(x, a), y).$
<u>Theories:</u>
$fold : \text{seq } E \times Acc \rightarrow Acc;$ $\forall y : Acc \bullet fold(\langle \rangle, y) = y,$ $\forall h : E, t : \text{seq } E, y : Acc \bullet fold(\text{append}(h, t), y) = fold(t, hd(y, h))$
<u>Types:</u>
Type List has specification: $\text{seq } E$
Type Element has specification: E
Type Acc has specification: Acc
<u>Fragments:</u>
<p>Fragment <code>processList(x:List)</code> has specification: output $y:Acc$ such that $y = f(x)$ implementation: return <code>accumulator(x,base)</code>.</p> <p>Fragment <code>accumulator(x:List,y:Acc)</code> has specification: output $z:Acc$ such that $z = fold(x, y)$ implementation: cases <code>decomposeList(x)</code> of: empty: return <code>y</code> nonempty: assign output to $v:Element, w:List$; assign <code>processElem(v, y)</code> to $y:Acc$; return <code>accumulator(w, y)</code></p> <p>Fragment <code>processElem(e:Element, a:Acc)</code> has specification: output $r:Acc$ such that $r = hd(e, a)$.</p> <p>Fragment <code>base()</code> has specification: output $b:Acc$ such that $b = base$.</p> <p>Branching fragment <code>decomposeList(x:List)</code> has specification: if $x = \langle \rangle$ then report <code>empty</code> else report <code>nonempty</code> with output $h:Element, t:List$ such that $x = \text{append}(h, t)$.</p>

Figure 2.8: The Accumulator template

$$Elem \rightsquigarrow \mathbb{N}, \quad Acc \rightsquigarrow \mathbb{N}, \quad f(s) \rightsquigarrow sum(s)$$

The instantiation of *base* can be deduced from the first applicability condition, since the sum of the empty list is zero (from the definition of *sum*). The instantiation of *dh* can be similarly deduced from the second condition, since the sum of the list *append(h, t)* is *h* plus the sum of the list *t*. This leads to the following instantiation:

$$base \rightsquigarrow 0, \quad dh(x, a) \rightsquigarrow x + a$$

Since $Elem = Acc$ and *dh* is AC, conditions 3 and 4 are automatically satisfied upon setting

$$hd(a, x) \rightsquigarrow x + a$$

The algorithm can be completed by implementing `base` and `processElem`. \square

In this section the individual parts which make up the template will be described separately. A collection of templates used in the thesis is given in Appendix C, which are described in more detail in Section 2.7.

2.4.1 Overview

A template is uniquely identified by a template name:

[*TemplateName*]

A template consists of an identifier, a collection of formal parameters (for functions, relations and sets), applicability conditions, domain specific theories, types and fragments. These are explained in more detail below.

<i>Template</i>
<i>template_name</i> : <i>TemplateName</i>
<i>fparams</i> : \mathbb{F} <i>FunctionParamDecl</i>
<i>rparams</i> : \mathbb{F} <i>RelationParamDecl</i>
<i>tparams</i> : \mathbb{F} <i>SetParamDecl</i>
<i>applic_conds</i> : \mathbb{F} <i>Assertion</i>
<i>declarations</i> : \mathbb{F} <i>OperatorDecl</i>
<i>assertions</i> : \mathbb{F} <i>Assertion</i>
<i>types</i> : \mathbb{F} <i>Type</i>
<i>fragments</i> : \mathbb{F} <i>Fragment</i>

2.4.2 Parameters

Formal parameters are mathematical constructs that can be instantiated to modify the meaning of a template. Parameters are represented by either a function name, relation name or set name. Associated with each function and relation parameter is a signature.

$$\begin{aligned} \text{FunctionParamDecl} &== \text{FunctionParam} \rightsquigarrow \text{FunctionSig} \\ \text{RelationParamDecl} &== \text{RelationParam} \rightsquigarrow \text{RelationSig} \\ \text{SetParamDecl} &== \text{SetParam} \end{aligned}$$

Fig. 2.9 shows the parameters for the accumulator template.

Name	Signature	Explanation
E	-	the type of the list elements
Acc	-	the type of the accumulated value
f	$\text{seq } Elem \rightarrow Acc$	the function to be computed
$base$	Acc	the initial value of the accumulator
hd	$Acc \times Elem \rightarrow Acc$	the function for processing successive elements
dh	$Elem \times Acc \rightarrow Acc$	an auxiliary function

Figure 2.9: Formal parameters for the accumulator template

2.4.3 Applicability conditions

To establish the correctness of an instantiated template, the user must discharge the instantiated applicability conditions. The applicability conditions provide sufficient conditions to establish the correctness of a template. The full correctness of a template (that is discharging any proof obligations associated with the template body) is established once off-line by a proof expert (typically the template designer). The applicability conditions capture any assumptions made about the parameters while proving the template. The applicability conditions for the accumulator template are given in Fig. 2.10.

$$\begin{aligned}
& f(\langle \rangle) = base, \\
& \forall h : E, t : \text{seq } E \bullet f(\text{append}(h, t)) = dh(h, f(t)), \\
& \forall x : E \bullet dh(x, base) = hd(base, x), \\
& \forall x, y : E; a : \text{Acc} \bullet dh(x, hd(a, y)) = hd(dh(x, a), y).
\end{aligned}$$

Figure 2.10: Applicability conditions for the accumulator template

2.4.4 Template body

The body of a template either encapsulates a commonly used refinement (data refinement or algorithmic refinement), provides a collection of primitive components or provides a problem specific domain theory. A template body consists of one or more fragments, types and theories. The *fold* function is defined in Fig. 2.11 which defines the well-known function for iteratively “folding” a function onto a list of values. Types which represent the list being processed, together with the type of the underlying list elements, and the type of the accumulated values are defined for the accumulator template in Fig. 2.12. The fragments in the body of the accumulator template are given in Fig. 2.13.

$$\begin{aligned}
& \text{Theory Definition of } fold. \\
& fold : \text{seq } E \times \text{Acc} \rightarrow \text{Acc}; \\
& \forall y : \text{Acc} \bullet fold(\langle \rangle, y) = y, \\
& \forall h : E, t : \text{seq } E, y : \text{Acc} \bullet fold(\text{append}(h, t), y) = fold(t, dh(h, y))
\end{aligned}$$

Figure 2.11: Theory for the accumulator template

$$\begin{aligned}
& \text{Type List has specification: } \text{seq } E \\
& \text{Type Element has specification: } E \\
& \text{Type Acc has specification: } \text{Acc}
\end{aligned}$$

Figure 2.12: Types for the accumulator template

2.4.5 Template verification

To establish the correctness of the template, all partial correctness, well-formedness, termination and non-execution proof obligations associated with the template body

```

Fragment processList(x>List) has
specification: output y:Acc such that  $y = f(x)$ 
implementation:
    return accumulator(x,base).

Fragment accumulator(x>List,y:Acc) has
specification: output z:Acc such that  $z = fold(x, y)$ 
implementation:
    cases decomposeList(x) of:
        empty:    return y
        nonempty: assign output to v:Element,w>List;
                 assign processElem(v,y) to y:Acc;
                 return accumulator(w,y)

Fragment processElem(e:Element,a:Acc) has
specification:
    output r:Acc such that  $r = hd(e, a)$ .

Fragment base() has
specification: output b:Acc such that  $b = base$ .

Branching fragment decomposeList(x>List) has
specification:
    if  $x = \langle \rangle$  then report empty
    else report nonempty
    with output h:Element,t>List such that  $x = append(h, t)$ .

```

Figure 2.13: Fragments for the accumulator template

must be discharged. All assumptions used to prove these proof obligations should either be part of the predefined theory, appear in the theory section of the template or be encapsulated in the applicability conditions. As mentioned earlier, the correctness of the template is established once, off-line by a proof expert. To establish the correctness of an instantiated template body, it is sufficient to discharge the template's applicability conditions. A proof of the accumulator template is given in a paper by Lindsay and Hemer [60], which also describes how the applicability conditions for this template are derived.

2.5 The CARE methodology

The CARE prototype toolset supports a simplified version of the well-established VDM approach to program development from formal specifications [9, 19, 48]. Unlike VDM and many other broad-spectrum languages, however, CARE uses a simple set of core constructs for algorithm and data refinement which means that the notation is easy to learn and easy to apply; this makes it a good subject for illustrating the ideas presented in this thesis. In brief, the major steps in the use of the CARE toolset are:

Domain theory: The software engineer begins by defining a mathematical theory which characterises the problem domain within which the application is to be developed. The theory typically consists of mathematical definitions of the types of object to be considered, together with functions on those objects and relationships between them. The CARE toolset accepts definitions written in a mathematical notation based on the Z “mathematical toolkit” which in turn is based on many-sorted set theory [69, 87]. A large user-extensible library of theorems is supplied with the toolset, including theories of commonly used mathematical constructs such as sets, sequences, relations and mappings.

Program specification: The next step is to formally specify the application program, by defining the desired relationship between its inputs and outputs. The data types to be used by the program also need formal specifications, in the form of mathematical expressions which define their carrier sets (that is, the set of values which belong to the type). The specifications may be abstract, in the sense that they involve mathematical concepts which are not immediately implementable in code. They may for example be defined implicitly or in terms of properties which are required to be kept invariant. In particular, CARE specifications are not necessarily executable [35].

Program development: Using CARE, the software engineer typically develops a program design by progressively adding algorithmic detail and refining abstract data structures into more concrete representations. The CARE toolset

includes a user-extensible library of generic design templates which record useful algorithm and data refinements. Program development eventually bottoms out when so-called “primitive” components are reached, which correspond to library routines and which are written directly in the target language. Note that the CARE user does not write target code.

Design verification: At any stage the correctness of the partial design can be checked by using the CARE tools to generate proof obligations which check that the components fit together properly and achieve the desired effect. Proof obligations are written as mathematical formulae whose truth should be judged before proceeding further with the design. In the first instance, the truth of the proof obligations should be judged informally by the software engineer, who needs to convince him or herself that they are logical consequences of the domain theory. If the proof obligations cannot be discharged it could be because the design is incorrect or the domain theory is incomplete.

Code synthesis: When the program design is complete, another CARE tool is used to automatically synthesise a source-code program with the same structure as the CARE program and with the target code from the primitive components included. If all the proof obligations can be discharged, then the synthesised program is guaranteed to meet its formal specification. (This assumes of course that the primitive components are correct: that is, that their mathematical specifications accurately characterise the target-code segments they contain.) The prototype toolset produces compilable C code, but in principle the approach could be adapted to produce code in most of the commonly used programming languages.

Program documentation: The CARE toolset includes pretty-printers and \LaTeX macros which allow CARE programs to be formatted for inclusion directly into \LaTeX documents.

CARE differs from most other formal software development methods by supporting incremental working – top-down, bottom-up or in a mixture of styles.

2.6 The CARE tools

This section describes in more detail the functionality of the individual tools in the CARE toolset. Most of the tools, with the exception of the theorem provers and the code synthesiser, were themselves formally specified in Z (see e.g. [39]) and implemented by an almost mechanical translation to Prolog. The overall architecture of the toolset is shown in Fig. 2.14.

Note that the following overview of the tool architecture is a snapshot of the toolset prior to the beginning of the work reported in this thesis. Certain enhancements have been made during the course of this project, in particular the addition of a retrieval tool (see Chapter 9), together with enhancements to the template instantiator (now referred to as the *template adaptor* in line with the fact that it now does more than instantiating parameters - see Chapter 4 for more details of these extensions). A more detailed description of the CARE toolset is given in a paper by Hemer and Lindsay [40].

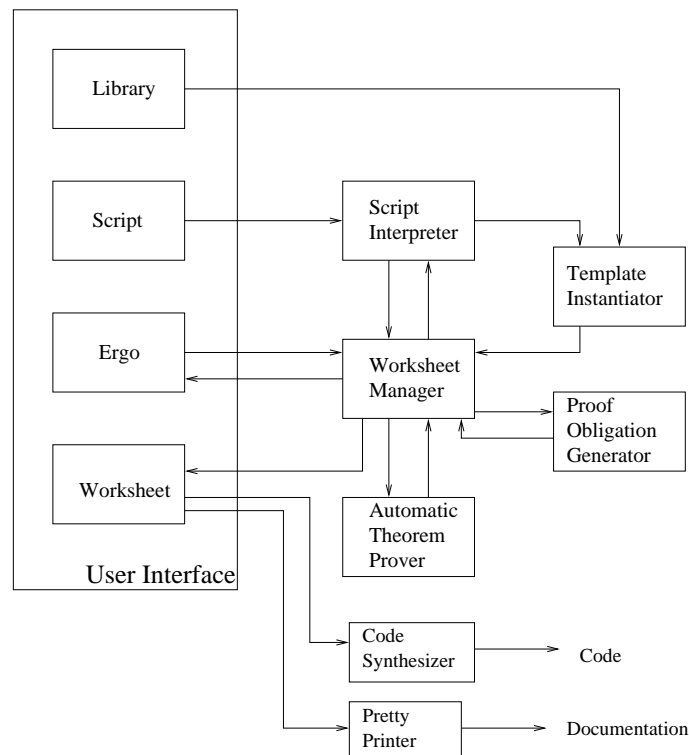


Figure 2.14: Architecture of the CARE toolset

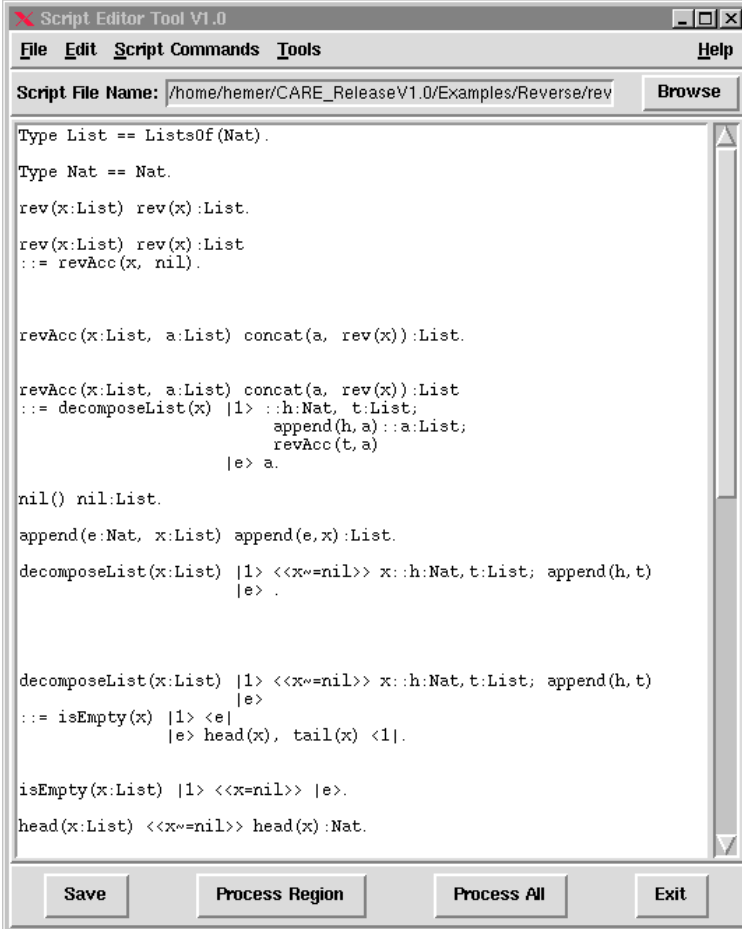
Script: The development and verification of a CARE program is driven from a script supplied by the software engineer. A script may include declarations of fragments, types and theories, as well as commands for retrieving and instantiating templates from the library, generating proof obligations, and invoking one of the theorem provers on a given proof obligation.

Worksheet: The current state of the CARE program under development is stored and displayed on a “worksheet”. The worksheet displays the fragments, types and theories that have either been written by the software engineer or gathered from the library, together with all the proof obligations that have been generated. Each component of the worksheet has an associated **status** which indicates the component’s standing in the overall development. For fragments and types, the status is one of the following: specified only; pre-proven (the component’s implementation comes from a library template); implemented but proof obligations not yet generated; proof obligations generated but awaiting proof; proven. For proof obligations, the status is either proven or unproven. Finally, the worksheet itself is considered complete and correct if and only if all its fragments and types are implemented (i.e., have status ‘pre-proven’ or ‘proof obligations generated’) and all associated proof obligations have been generated and discharged. Note that worksheets are not directly editable by the software engineer: information can only be added to the worksheet or modified via the script.

Library: The library consists of a collection of pre-proven design templates (see Section 2.4 for more details).

Script interpreter: The script interpreter parses the individual script commands and passes annotated fragments, types and theories to the worksheet manager as abstract syntax trees. An example of the script interpreter being used in the development of a program for reversing a list is given in Fig. 2.15. Note that the concrete syntax for the CARE language used by the tools, is slightly different to that presented in this thesis. In particular the tools use an

ASCII representation of the syntax, which is terser than the concrete syntax presented in the thesis. Other differences in the concrete syntax are described in Chapter 9.



```

Script Editor Tool V1.0
File Edit Script Commands Tools Help
Script File Name: /home/hemer/CARE_ReleaseV1.0/Examples/Reverse/rev Browse

Type List == ListsOf(Nat).
Type Nat == Nat.
rev(x>List) rev(x):List.
rev(x>List) rev(x):List
::= revAcc(x, nil).

revAcc(x>List, a>List) concat(a, rev(x)):List.

revAcc(x>List, a>List) concat(a, rev(x)):List
::= decomposeList(x) |1> ::h:Nat, t>List;
      append(h, a)::a>List;
      revAcc(t, a)
      |e> a.

nil() nil>List.
append(e:Nat, x>List) append(e, x):List.
decomposeList(x>List) |1> <<x=nil>> x::h:Nat, t>List; append(h, t)
      |e> .

decomposeList(x>List) |1> <<x=nil>> x::h:Nat, t>List; append(h, t)
      |e>
::= isEmpty(x) |1> <e|
      |e> head(x), tail(x) <1|.

isEmpty(x>List) |1> <<x=nil>> |e>.
head(x>List) <<x=nil>> head(x):Nat.

Save Process Region Process All Exit

```

Figure 2.15: The script interpreter for the reverse example

Template instantiator: This tool is given a template name and an instantiation of its formal parameters. It then retrieves and instantiates the template appropriately, and passes the results to the worksheet manager. The functionality of this tool has been extended as described later in the thesis.

Proof obligation generator: Proof obligations are generated purely mechanically from the CARE components and simplified using basic properties of equality, propositional calculus and quantifiers.

Theorem provers: The CARE toolset includes two theorem provers, one fully automatic and the other interactive (the Ergo theorem prover). Both theorem provers are more or less stand-alone tools.

Worksheet manager: The worksheet manager controls what goes on the worksheet, where it is placed on the worksheet and with what status. It takes its input from the script interpreter and from the theorem provers, and updates the worksheet accordingly. The worksheet manager is responsible for reporting various errors back to the user via the script interpreter, for example if the user tries to overwrite an already existing implementation. Fig. 2.16 shows the state of the worksheet after the script commands given in Fig. 2.15 have been processed by the script interpreter and worksheet manager.

Pretty printers: A number of pretty printers are available to convert the worksheet from its abstract syntax tree form into human readable forms, for example in ASCII (suitable for reparsing) or \LaTeX form (for inclusion in printed documents).

Code synthesiser: The code synthesiser tool takes a complete collection of fragments and types and constructs a C source-code program. In the initial phase of code synthesis, a code graph is constructed for each higher-level fragment in the collection. In the second phase, a series of transformations expands the graph for a user-nominated “main” fragment until a single code graph is obtained in which all the (non-recursive) fragments have been fully expanded. The final step produces the synthesised program by translating the code graph into appropriate code in the target language, drawing in the code fragments associated with primitive components and performing various optimisations along the way. The reader is referred to [64] for further details on the code synthesis process.

```

Worksheet Viewer Tool V1.0
File View Text Show Tools Help

Worksheet View

Type List==ListsOf(Nat)::=(v)l:"List l" Assign ::= l Associated code "
struct linked_list { Nat val; struct linked_list * next;};", "typedef
struct linked_list POINTER;";
"typedef POINTER * List;".

Type Nat::=(v)<<v <= uintmax>>i:"Nat i" Assign ::= i Associated code
"typedef unsigned int Nat;".

rev(x>List)  rev(x)::l0>List;l0
             ::= revAcc(x,nil)::l0>List;l0 .

revAcc(x>List, a>List)  concat(a, rev(x))::l1>List;l1
                       ::= decomposeList(x) |l> ::h:Nat, t>List;append(h, a)::a0>List;
revAcc(t, a0)::l0>List;l0 |e> a::l0>List;l0 .

nil()  nil::l1>List;l1
       ::= "Null"::v1:"List v1";v1 -> v>List;v::l1>List;l1 .

append(e:Nat, x>List)  append(e, x)::l2>List;l2
                      ::= e <: eval:"Nat eval";x <: v1:"List v1";"malloc(sizeof(POI
NTER))"::v11:"List v11";"v11->val = eval"(v11, eval::v11:"List v11");"v
11->next = v1"(v11, v1::v11:"List v11");v11 -> v1>List;v1::l0>List;l0 .

decomposeList(x>List)  |l> <<x ~= nil>>x::h:Nat, t>List;append(h, t) |e>

Proof obligations

Partial Correctness Proof Obligation rev_partial_1
All x:ListsOf(Nat). rev(x) = concat(nil, rev(x))
status unproven.

Partial Correctness Proof Obligation revAcc_partial_1
All x:ListsOf(Nat), a:ListsOf(Nat). x ~= nil => (All h:Nat, t:ListsOf(Nat)
). x = append(h, t) => concat(a, rev(x)) = l1
status unproven.

Partial Correctness Proof Obligation revAcc_partial_2
All x:ListsOf(Nat), a:ListsOf(Nat). x ~= nil => concat(a, rev(x)) = l1
status unproven.

Partial Correctness Proof Obligation decomposeList_partial_1

reverse.wc                                RCS Version 1.7
Reload Undo                                Exit

```

Figure 2.16: The worksheet manager for the reverse example

2.7 Library

This section describes the templates used in this thesis. A complete listing for each of these templates is given in Appendix C.

2.7.1 Algorithm refinements

A commonly used list processing algorithm is list *accumulation*. Here a list is processed in a step-wise manner, with an intermediate value being accumulated at each stage. We present two templates for list accumulation in this thesis. The first of these, the so-called standard accumulator template is given in Fig C.1, and has already been described in some detail earlier in this chapter.

A special case of the standard accumulator is the *associative commutative accumulator* template, given in Fig. C.2. This template is much the same as the standard accumulator, except that the step function is assumed to be associative and commutative (AC) which results in simplifications to the algorithm, in particular the auxiliary function is known to be the same as the step function, and the applicability conditions are generally simpler. The applicability conditions for this template are restated in Fig 2.17. The first two applicability conditions are the same as those for the standard accumulator template. The third and fourth capture the assumption that the step function is commutative and associative respectively.

$$\begin{aligned}
 & f(\langle \rangle) = base \\
 & \forall h : E, t : \text{seq } E \bullet f(\text{append}(h, t)) = hd(f(t), h) \\
 & \forall x, y : E \bullet hd(x, y) = hd(y, x) \\
 & \forall x, y, z : E \bullet hd(x, hd(y, z)) = hd(hd(x, y), z)
 \end{aligned}$$

Figure 2.17: Applicability conditions for the associative commutative accumulator

The `if then intro` template given in Fig. C.3 on page 273 implements a binary (simple) fragment by introducing an exception case and treating this separately to the “normal” behaviour. If a test on the inputs succeeds (indicating the exception case), the second argument is returned; otherwise a result, which is a function of the inputs, is returned.

2.7.2 Data refinements

In Appendix C, three templates containing refinements of sets in terms of lists are given. The first of these, sets in terms of (possibly repeating) *lists* is given in Fig. C.6. For this particular refinement, the set is represented by the range of the list. For example $\langle a, b, a, a, c \rangle$ and $\langle c, a, b \rangle$ both represent the set $\{a, b, c\}$. Note that no invariant is given for this data refinement; in particular repetitions are allowed in the list. The data refinement for sets as repeating lists is repeated in Fig 2.18.

The second of the data refinements is sets as *non-repeating lists* (see Fig. C.7). The data refinement repeated in Fig. 2.19, shows that like the previous template,

Type **Set** has specification: $\mathbb{F} E$
 refinement:
 value **s** is refined by **l:List**
 with refinement relation $s = \text{ran } l$.

Figure 2.18: Refining sets as (possibly) repeating lists

the refinement relation indicates that the set is represented by the range of the list. However in this case an invariant is given stating that the list representing the set cannot contain any repetitions. For example the list $\langle a, b, a, a, c \rangle$ used to represent the set $\{a, b, c\}$ for the previous example cannot be used in this case.

Type **Set** has specification: $\mathbb{F} E$
 refinement:
 value **s** is refined by **l:List** with invariant $IsNonRep(l)$
 with refinement relation $s = \text{ran } l$.

Figure 2.19: Refining sets in terms of non-repeating lists

The third of the three data refinements is sets in terms of *ordered lists* (see Fig. C.8). For this template there must be some ordering on the elements which make up the set. The data refinement (see Fig. 2.20), has the same refinement relation as for the previous two templates, but in this case the underlying list must be ordered. For example the ordered list $\langle 1, 3, 17 \rangle$ represents the set $\{1, 3, 17\}$.

Type **Set** has specification: $\mathbb{F} E$
 refinement:
 value **s** is refined by **l:List** with invariant $IsOrdered(l)$
 with refinement relation $s = \text{ran } l$.

Figure 2.20: Refining sets in terms of ordered lists

Each of these three templates contains two fragments for manipulating sets: one for performing binary operations on sets (e.g. union, intersection, difference etc.) and one for performing operations on a set and an element to give another set (e.g. adjoining an element to a set, removing an element from a set, etc.). These set manipulation fragments are implemented in terms of fragments on lists, with appropriate abstractions and representations performed.

The three sets as lists refinements have relative advantages and disadvantages. For sets as repeating lists, adding a new element is an efficient operation, since no check for repetitions is required, nor is there any need to put the element in a specific place. However searching for an element in the underlying list will be inefficient in general, since a linear search is required, with the possibility of the search space being increased by repetitions. Also with repetitions, the refinement can be uneconomical with memory.

In contrast, for sets as non-repeating lists, adding a new element is less efficient, since a check for repetitions needs to be done. Searching for an element in this case is in general more efficient since the search space will be smaller (no repetitions), but a linear search is still required. The refinement makes optimal use of memory space, with no repetitions of elements allowed.

For sets as ordered lists, adding new elements to the set can be inefficient because elements need to be inserted within the list in line with the ordering, which in general is far harder than just appending it to either end of the list. Searching the list for elements is far more efficient than for the other two refinements, because search algorithms such as binary search can be employed which in general are more efficient than a linear search.

2.7.3 Primitive components

The third class of templates in Appendix C are the *primitive component* templates. These templates contain fragments and types implemented directly in target language code, in this case in C code. Typically a primitive component template will be the software engineer's only access to target code constructs, since they will not normally be allowed to develop primitive components themselves. Primitive component templates will normally consist of one or more primitive type implementations, together with a number of primitive fragments which manipulate objects of this type.

The natural numbers as unsigned integers template given in Fig C.5, contains primitives for representing natural numbers as unsigned integers in C. A number of

fragments which perform basic operations on these data structures are included in the template: included are addition, subtraction, multiplication, division, equality and inequality tests, together with fragments for implementing the constants “0” and “1”.

A template for representing sequences as (singly) linked lists in C is given in Fig. C.4. The template is parameterised over the type of the underlying list elements, and can be instantiated to give lists of different types. Primitive components are given for: appending elements to the list, calculating the head and tail of a list, representing the empty list, and testing whether a list is empty.

2.8 Discussion

The abstract syntax for the CARE language given here includes most of the different kinds of constructs found in formal languages. Indeed most meta-language features are covered including: variables, functions, formulae, binders, applications, parameters and typing.

The unit level includes function-like operations (called fragments), data types, assertions and operator declarations. Language features covered by units are: inputs and outputs (and their types); preconditions and postconditions; separation of specifications and implementations; textual and formal parameters; and case statements.

Finally module-like constructs are covered in the abstract syntax in the form of templates. Other features covered by templates include: specified-only and implemented units; applicability conditions; encapsulation; and information hiding. As a result of the broad coverage of the CARE language, what follows in this thesis should be adaptable to a wide range of formal languages.