SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-9

The Care method of verified software development

Peter A. Lindsay

June 1995

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

The CARE method of verified software development *

Peter A. Lindsay,
Software Verification Research Centre
email: pal@cs.uq.edu.au.

Abstract

This paper describes the CARE method for developing formally verified software. The method partitions the software development task in such a way that formal verification takes place in parallel with design and implementation. The aim is to separate the software design aspects of verified software development from the formal mathematical aspects.

This paper presents the conceptual basis for the CARE method of algorithm development from formal program specifications. The CARE notation is introduced, together with a process of proof obligation generation. The method is illustrated on a small development.

Contents

1 Introduction **3**
1.1 Motivation 3
1.2 The CARE method 4
1.3 Verification under CARE 5
1.4 Using CARE in system development 6
1.5 This paper 8

2 Preliminaries **9**
2.1 Mathematical notation 9
2.2 CARE types 9

3 Fragment specifications **10**
3.1 Simple and branching fragments 10
3.2 Specification notation 11

*The CARE project is a collaboration between the SVRC and Telectronics Pacing Systems Pty Ltd, supported by Generic Technology Grant No. 16038 from the Industry Research and Development Board of the Australian Government's Department of Industry, Science and Technology.

4 Fragment implementations **13**
4.1 Primitive fragments . . . 13
4.2 Simple fragments . . . 13
4.3 Recursion and variants . . . 14
4.4 Branching fragments . . . 15
4.5 Abort . . . 15
4.6 Auxiliary fragments . . . 16

5 Fragment verification **16**
5.1 Proof obligations . . . 16
5.2 Partial correctness . . . 17
5.3 Termination . . . 18
5.4 Well-formedness . . . 19
5.5 Non-execution . . . 19

6 An example development **20**
6.1 Program specification . . . 20
6.2 First design step . . . 20
6.3 Verification of first design step . . . 21
6.4 Second design step . . . 22
6.5 Third design step . . . 23

7 An example programming technique **24**
7.1 Templates . . . 24
7.2 An accumulator . . . 24
7.3 A template for accumulators . . . 25
7.4 Example: summing a list . . . 27
7.5 Example: reverse a list . . . 27
7.6 Example: list maximum . . . 28
7.7 Verification of the template . . . 28

8 Conclusions **30**

1 Introduction

1.1 Motivation

Computer software is increasingly used to control systems whose malfunction may threaten life, compromise national security, or have other serious consequences. Formal (mathematically-based) techniques of software development are proposed as a means of significantly enhancing the nature of assurance in the correctness of the delivered software. The emerging importance of formal methods is increasingly being recognised by government regulatory and standards authorities, as evidenced by the growing number of standards which mandate or recommend the use of such methods in the development of trusted software [1, 2, 3, 4, 5].

Formal specification techniques offer improved understanding of the system being developed and present opportunites to perform cross-checks on specifications, thereby discovering and fixing mistakes early in the development life-cycle. Formal specification techniques are already being used widely [6, 25, 30].

Formal development techniques extend the benefits of formal specification further into the development process by allowing the user to express design and implementation strategies for meeting formally specified requirements. Formal verification can then be used to check the correctness and completeness of those strategies, perhaps revealing gaps that need to be plugged. As currently practised, however, formal software development is a laborious and time-consuming task, calling for specialized mathematical skills.

Formal verification of a medium-sized application typically would include the following tasks. Each software component is formally specified and implemented, and the implementation is shown to satisfy the specification. Each software module, consisting of a collection of individual software components, is formally specified and the combination of components is shown to satisfy the module specification. Interfaces between software modules — and between software and other system modules (hardware devices, users, off-the-shelf products, etc.) — are formally specified, and the modules are shown to respect the interfaces. The whole design is shown to meet its functional specification as well, perhaps, as satisfying certain other high-level properties, such as having certain safety or security features.

In principle, formal development can be carried all the way through to code. In practice however very few target implementation languages have complete formal semantics. As a result, most formal development methods stop at the level of programs expressed in a formally-defined intermediate language. Programs are then ported to the desired target language.

The most cost-effective time to establish correctness is during development when design decisions are being made and checked, rather than afterwards during testing as is done when using traditional development techniques. After-the-fact verification is generally not feasible: correctness needs to be designed into a product. For these reasons it is important that formal development techniques be usable by non-mathematicians.

A method is needed which structures the development process in such a way that verification steps are individually manageable by the software developer. Such an

approach has shown to be successful when the design space is tightly constrained, for example in the AMPHION system [28] where programs are derived from specifications written by space scientists, using a library of formally-specified Fortran routines. In AMPHION's case, proof obligations are discharged by a mechanical theorem prover behind the scenes.

1.2 The CARE method

This paper describes the CARE approach to developing formally verified software. CARE stands for **Computer Assisted Refinement Engineering**. The aim of the CARE method is to partition the software development task in such a way that formal verification takes place in parallel with design and implementation, as unobtrusively as possible. The aim is to separate the "engineering" aspects of verified software development (such as requirements specification, algorithm design, and choice of data structures) from the "scientific" aspects (such as mathematical modelling, proof obligation generation, and formal proof of correctness). As far as possible, formal verification aspects of the method are consigned to automated tools, allowing the software engineer to concentrate instead on the design and development of usable, efficient pieces of software.

The CARE language is used to express program specifications and designs. Special-purpose tools produce code in a compilable target language, together with a certificate of correctness for the code (see Fig. 1). The method is essentially independent of the particular specification notation used, and is largely target language/compiler/platform independent.

Under CARE, the programming/verification task is structured in a natural and effective manner, whereby programming knowledge is packaged into reusable components called *types* and *fragments*. Roughly speaking, a CARE type is an abstract data structure, and a CARE fragment is a package of programming knowledge corresponding roughly to a function or procedure in a procedural programming language.

Each CARE type and fragment has a formal specification and an implementation. *Primitive* types and fragments provide access to target language data structures and basic functionality, and are provided to the CARE user as a library. Primitive types and fragments are implemented directly in the target language. The formal specification of a primitive type or fragment describes it mathematically.

Higher-level types and fragments express data refinements and algorithm designs, and are written in a special purpose language. The CARE language supports simple design constructs such as assignment of values to local variables, fragment calls, sequencing, branching of control, recursion, and data refinement. The specification part of the language supports many-sorted predicate calculus. CARE components may have associated applicability conditions which define the circumstances under which the component may be applied.

The CARE language has a formally-defined mathematical semantics. Using this semantics, higher-level components can be shown to be correctly implemented, assuming the subcomponents they use have themselves been correctly implemented. Proof obligations, generated mechanically from the components' definitions, check that appli-

cability conditions are satisfied and that implementations achieve their specifications. When a component set is complete and the proof obligations have been discharged, a target language-specific CARE tool synthesizes a complete source code program from the set. The CARE process is summarized in Fig. 1.

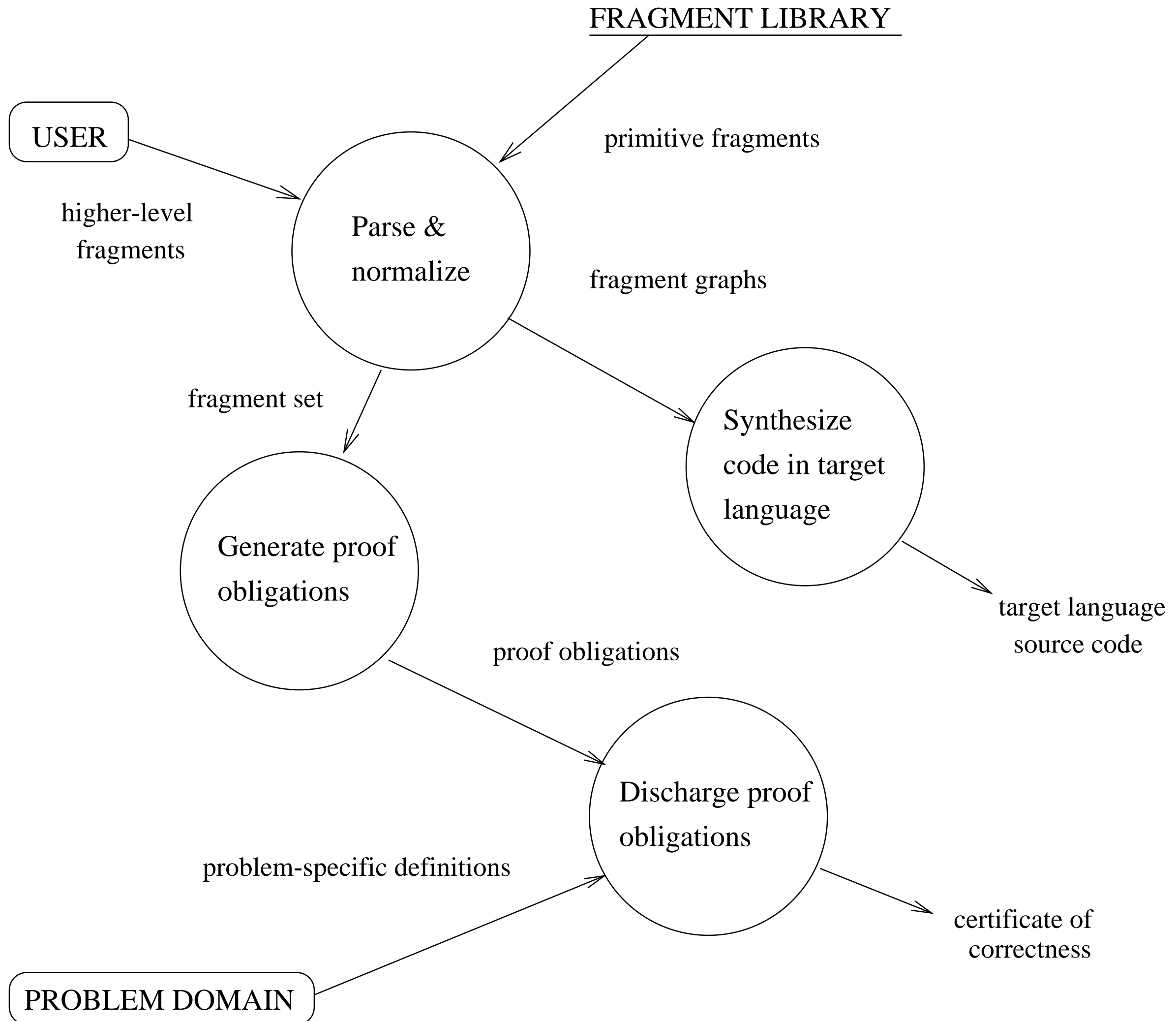

Figure 1: The CARE process of verified software production.

1.3 Verification under CARE

The CARE method is designed to allow much of the verification task to be done off-line, in reusable fragments, independent of other software development tasks. The main aspects of the approach are:

1. The correctness of each higher-level fragment is verified in isolation as far as possible, using information from the specifications (only) of the fragments they call.[1]

[1] Verification of termination, however, generally requires full details of the set of mutually recursive fragments.

2. The choice of design constructs supported by the CARE language strikes a balance between design flexibility and ease of verification.

3. The interface to the target language is via primitive types and primitive fragments, taken from a library. The correctness of primitive components is established off-line, by techniques appropriate to the target language in question (not part of the CARE method).

4. The CARE language is typed and type-checking is fully automatable. This takes some of the load off the theorem prover by revealing certain classes of semantic error earlier. It also allows the theorem prover to use type information in its reasoning.

5. Designs can be expressed largely independently of the target language, which means they are more easily portable. [2]

6. Common program refinement techniques can be expressed as parameterized fragment *templates* and verified off-line for later reuse: see Section 7 for an example.

Because the logical relationship between specification and implementation is maintained by the method, the code produced by the method is fully traceable back to the program specification, and vice-versa. Software designs can be modified by changing individual fragments, and then rerunning the tools on the whole design to generate a new certificate of correctness and a new target language program. Components of the final design which are platform — or language — specific are isolated in primitive fragments, for separate verification. The sum effect of this approach is that the design aspects of software development are separated out from the mathematical aspects of software verification in a practical and effective manner.

1.4 Using CARE in system development

CARE is a method for developing verified software from formal program specifications. The method itself is largely independent of the particular specification language employed: this paper uses the Z notation [27] since it is widely familiar, but the method could be tailored to fit with other formal specification notations, such as VDM-SL [10] or the Larch Shared Language [13]. The CARE language currently supports applicative programming techniques only, but there are plans to extend this to cover fragments which can change the value of a hidden state.

For software system development, we imagine that CARE would be used in conjunction with a method or methods for requirements capture, system specification and system design, where such a method results in program module specifications: see [22] for an example. CARE is most effective where there is a requirement for formally verified software, such as when dealing with high integrity software components, or when the logic of the program to be developed is complex or unfamiliar. Because they are target language source-code programs, CARE-synthesized programs can be integrated

[2] See e.g. the Larch literature [13] for a discussion of the importance of separating design considerations from target language considerations.

with other system components and tested using traditional integration techniques: see Fig. 2. Also, Care can be used to produce programs in target languages which do not have fully formally-defined semantics, by restricting primitive fragments to pieces of target language code which have a mathematically-definable meaning.

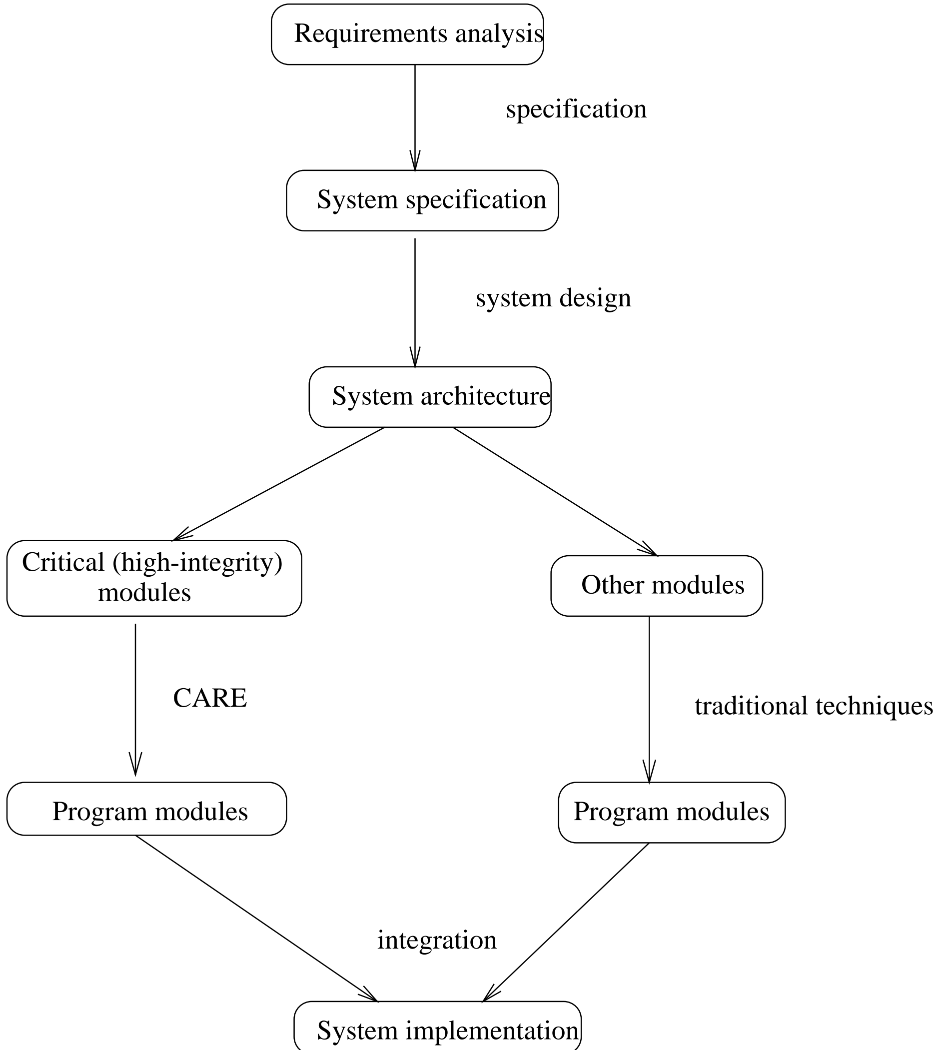

Figure 2: Use of Care in system development using the traditional waterfall model.

Care can also be used in conjunction with formal development techniques such as VDM [20] or the Refinement Calculus (RC) [24]. We are still investigating the relationship between the different techniques, but all available evidence seems to indicate that Care developments can easily be translated into VDM or RC developments/refinements and vice-versa: see [21] for the translation from RC to Care. In part, Care can be seen as a way of giving further structure to VDM or RC developments — structure which is useful for raising the level at which one reasons about designs and design choices, while hiding lower-level verification aspects. We plan to further investigate a closer integration of these different approaches.

The Care method is informed — and partly inspired [26] — by the Verification Condition Generation (VCG) approach to program verification [11, 12], and could be combined with such approaches. In particular, we have purposefully left open the range of

techniques used for verification of primitive (target-language implemented) types and fragments.

We have also left open the kinds of theorem prover that could be used with CARE. Our project is currently exploring two different technologies. Keith Harwood and his team at Telectronics have developed a purpose-built automatic theorem prover based on many-sorted resolution under equality. At the SVRC we have adapted the Ergo interactive proof assistant [29] by using its store of theorems about many-sorted set theory and developing special-purpose simplification procedures and appropriate tactics and heuristics for the kinds of proof that arise under CARE. Since the CARE method is largely independent of the mathematical language used in specifications, it would be possible to adapt it — if desired — to other mathematical languages and other theorem provers, such as Boyer-Moore [8] or the EVES prover [11].

Finally, the Telectronics team are prototyping a set of tools for assisting the CARE user in constructing top-down developments by selecting and instantiating program templates (also known as *scenarios*).

1.5 This paper

This paper presents the conceptual basis for the CARE method of algorithm development. (CARE also covers data refinement [20] but the topic will not be treated here.) The CARE language has been evolving over a number of years [14, 15] and the notation has changed significantly during that time. This paper uses a verbose form of the CARE notation intended primarily for expository purposes: our prototype tools use a more concise notation more suited to machine processing.

The paper is structured as follows. Section 2 below outlines the mathematical notation used in the paper for expressing program specifications and defines the CARE types used. Section 3 explains the CARE notation for fragment specifications, illustrated with a number of examples. Section 4 describes the CARE language for implementing fragments. Section 5 discusses the semantics of the CARE language and outlines the proof obligations associated with a fragment collection. Section 6 gives an example of algorithm development in CARE. Section 7 illustrates how a programming technique can be expressed as a CARE template.

A companion paper [18] formalizes the process of generating the proof obligations associated with use of the CARE method. The report [23] describes code synthesis — the process of combining target code from primitives using higher-level types and fragments to guide the construction of a target-language program. As outlined above, the CARE method is largely target-language independent: use of target-language code is confined to primitive types and primitive fragments and will not be treated here. Case studies in CARE are presented in [22], and [19] is a formal specification of the CARE language.

2 Preliminaries

2.1 Mathematical notation

This paper uses Z [27] as the mathematical notation for writing program specifications since it is widely familiar. Z uses a set-theory based, many-sorted predicate calculus which has gained widespread usage. A recent U.K. survey [6] reported that, of the 400 respondents (70% from industry, 30% academic), over half currently use formal methods, and that of these, the two most widely used are Z and VDM. Industrial organisations which use Z include IBM, ICL, British Telecom, Hewlett Packard, British Aerospace, BP, Logica and Tektronix (USA). An international standard for Z is currently being prepared under the auspices of the International Standards Organisation (ISO) [9].

Z notation	meaning
$\mathbb{Z}$	the set of all integers
$\mathbb{N}$	the set of all natural numbers
$(m \,..\, n)$	the set of integers between m and n inclusive
$m \operatorname{div} n$	the integer part of m divided by n ($n \neq 0$)
$A \cup B$	the union of sets A and B
$A \rightarrow B$	the set of all total functions from set A to set B
$a \in A$	a is an element of set A
$max\ A$	the maximum element in a finite, non-empty set of integers A
$\operatorname{seq} X$	the set of all finite sequences over X
$\langle\,\rangle$	the empty sequence
$\langle e \rangle$	the singleton sequence consisting of e alone
$s \frown t$	the concatenation of sequences s and t
$\#s$	the length of sequence s
$s(i)$	the element at index i in sequence s (i.e., the ith element of s)
$head(s)$	the head (first element) of sequence s
$tail(s)$	the tail (i.e., all elements but the first) of sequence s
$last(s)$	the last element of sequence s
$ran(s)$	the range of (i.e., set of all elements in) sequence s
$rev(s)$	the reverse of sequence s

Table 1: The Z mathematical notation used in this paper.

Table 1 summarizes the Z notation used in this paper. In the body of the paper, Care values and types are written in `typewriter` font and mathematical expressions are written in *italics* using the Z notation.

2.2 Care types

The Care type system links target language data structures with their abstract mathematical counterparts. The specification of a Care type is an expression which denotes the set of mathematical values which the Care value can take (its "carrier set").

Like fragments, CARE types can be primitive or higher-level. Primitive types are implemented directly as target language data structures and are modelled as mathematical sets. Table 2 gives some examples of mathematical modelling of programming language data structures. Since this paper concentrates on the target language-independent aspects of CARE, details of the implementation of primitive types will not be given here.

Programming language data structure	corresponding mathematical set of values
arbitrary precision integers	$\mathbb{Z}$
10-bit integers	$(-511 \,..\, 512)$
arbitrary-length strings of characters	$\mathrm{seq}\, CHARACTERS$
fixed-size arrays, of size k, of elements	$(1 \,..\, k) \rightarrow ELEMENTS$
linked lists of elements	$\mathrm{seq}\, ELEMENTS$

Table 2: Examples of modelling of data structures in Z.

Higher-level CARE types are used for expressing data refinements: that is, changes of mathematical representation of data types (called data reification in [20]). For example: sets might be implemented in terms of non-repeating sequences; and rational numbers might be implemented as pairs of integers. The case study [22] illustrates the use of data refinement in CARE. The implementation in CARE of higher-level types will not be discussed further in this paper.

Table 3 defines the CARE types used in this paper.

CARE type	data structure	mathematical set
`Integer`	integers	$\mathbb{Z}$
`Natnum`	natural numbers	$\mathbb{N}$
`Element`	elements	$ELEMENTS$
`List`	lists of elements	$\mathrm{seq}\, ELEMENTS$
`NatList`	lists of numbers	$\mathrm{seq}\, \mathbb{N}$

Table 3: CARE types used in this paper.

3 Fragment specifications

3.1 Simple and branching fragments

There are two kinds of fragment: simple and branching.

A *simple fragment* takes inputs and produces outputs. The number and type of inputs taken by a fragment is fixed. In addition, there may be a *precondition* (or applicability condition) which further limits the inputs that can be supplied to the fragment: for example, a fragment for finding the head of a list may require that the list be non-empty. There will be proof obligations to show that the precondition is satisfied each time the fragment is called.

Like a simple fragment, a *branching fragment* takes inputs and produces outputs. The number and type of inputs is fixed, and there may be a precondition. Unlike a simple fragment, however, the number and type of outputs of a branching fragment may be different for different inputs. For example, a branching fragment for "decomposing" a list may have two different cases: one for when the list is empty (in which case it returns no outputs), and the other for when it is non-empty (in which case it would return the head and tail of the list). The different cases are distinguished by formulae called *guards*. A branching fragment produces a *report* to indicate which case has arisen.

3.2 Specification notation

A fragment's *specification* describes the fragment mathematically. The specification defines the types of the inputs, the precondition, the different results (reports and output types) that may arise, and the required *input/output (I/O) relationship* in each of the cases. These notions are illustrated on examples below.

Fragment specifications may be underdetermined, in the sense that more than one output may satisfy the required I/O relationship for any given input. This degree of nondeterminism is useful for writing abstract specifications and for allowing detailed design choices to be postponed until later in the development process [17]. This is illustrated by an example in Section 6 below.

Example (1) The specification of a simple fragment for finding the head of a non-empty list:

> Fragment `car(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `h:Element` such that $h = head(s)$.

The name of the fragment is `car`. (The reason for the LISP-like naming will become apparent as the paper progresses.) The fragment has a single input parameter `s` of type `List`. The fragment's precondition is that the input be non-empty: this is expressed as $\#s \neq 0$, where s is the mathematical value corresponding to the CARE value `s`. (This convention applies to all CARE variables: `typewriter` font is used for the CARE value and *italics* for the corresponding mathematical value.) The fragment's output `h` is of type `Element` and is defined in terms of the relationship it has to the original list: in this case, $h = head(s)$.

Example (2) The specification of a simple fragment for finding the tail of a non-empty list:

> Fragment `cdr(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `t:List` such that $t = tail(s)$.

Example (3) The specification of a simple fragment for the empty list:

Fragment `nil` has
specification:
output `s:List` such that $\#s = 0$.

(Where the precondition is 'true' it is omitted.)

Example (4) The specification of a simple fragment for putting an element onto the front of a list:[3]

Fragment `cons(e:Element,s:List)` has
specification:
output `r:List` such that $r = \langle e \rangle \frown s$.

Example (5) The specification of a branching fragment for checking whether a list is empty:

Branching fragment `null(s:List)` has
specification:
result defined by cases:
if $\#s = 0$ then report `yes`
else report `no`.

The `null` fragment has two branches. The first branch, with guard $\#s = 0$, simply reports `yes` without returning any outputs: this branch is taken when s is empty. The second branch is taken when s is not empty: in this case, the fragment reports `no`, again with no outputs.

Example (6) The specification of a 3-way branching fragment for comparing two integers:

Branching fragment `compare(m:Integer,n:Integer)` has
specification:
result defined by cases:
if $m < n$ then report `lessthan`
elseif $m = n$ then report `equal`
else report `gtrthan`.

For this example, the guards on the three branches are $m < n$, $m = n$ and true, respectively. (As usual, the 'true' is omitted.) As the notation suggests, the guards are evaluated in turn to determine which branch should be taken: the first branch is taken if m is less than n, the second if m is equal to n, and the third branch otherwise.

Example (7) The specification of a branching fragment which searches a list for the index (if any) of a given element:

Branching fragment `search(s:List,x:Element)` has
specification:

[3] An equivalent I/O relationship is $r \neq \langle\rangle \wedge \mathit{head}\, r = e \wedge \mathit{tail}\, r = s$.

result defined by cases:
  if $x \in ran(s)$ then report `found`
    with output `i:Natnum` such that $s(i) = x$
  else report `notfound`.

The `search(s,x)` fragment has two cases: when `x` occurs in `s`, it reports `found` and returns an index at which `x` can be found; otherwise it simply reports `notfound`. Note that the specification is underdetermined: for example, if `x` occurs more than once in `s`, the specification does not uniquely define which index `i` should be returned.

4 Fragment implementations

4.1 Primitive fragments

Primitive fragments are ones whose implementation involves target language code. Since this paper concentrates on the target language-independent aspects of CARE, details of the implementation of primitive fragments will not be given here.

A primitive fragment's specification describes the associated target code's functionality in terms of (a mathematical model of) the semantics of the target language and its compiler. For example, a procedure which appends a new cell onto a linked list might be modelled mathematically as a function which conjoins an element onto a finite sequence.

Primitive fragments are written by a trusted party — a specialist in mathematical modelling of target language code — and verified outside the CARE method using techniques appropriate for the particular target language. The CARE method simply assumes that primitive fragment specifications are correct: that is, that they accurately capture the semantics of the fragment's target code.

The ordinary CARE user does not write primitive fragments: they can only use the ones supplied with the CARE system. At the same time, the only access the CARE user has to the target language is via primitive fragments. In this way, primitive fragments form a kind of firewall, preventing the ordinary user from interacting directly with the target language. Over time, we anticipate that large libraries of reusable primitive fragments will be developed by verification experts. The correctness of individual fragments would be checked by rigorous peer review, and only those fragments which pass the most stringent reviews would pass into general use.

The rest of this section describes the CARE language for constructing higher-level fragments.

4.2 Simple fragments

Higher-level fragments are implemented in the CARE language, which is tree-structured. Non-branching nodes of the tree correspond to bindings to local variables of the values returned by simple fragment calls and/or variables. Branching nodes correspond to calls to branching fragments; where branches return values, these values get bound to

local variables. The leaves of the tree define the fragment's outputs — as well as the report, in the case of branching fragments. These ideas are illustrated on examples below using fragments whose specifications are given in Section 3.2 above.

Here is a simple fragment which takes an element and constructs a singleton list from it:

> Fragment `makeList(e:Element)` has
> specification:
>   output `s:List` such that $s = \langle e \rangle$.
> implementation:
>   assign `nil` to `t:List`;
>   return `cons(e,t)`.

For this example, the tree has no branching. Execution of `makeList(e)` assigns the value of the `nil` fragment (the empty list) to local variable `t`, calls the `cons` fragment on arguments `e` and `t`, and returns the result. A proof obligation (see Section 5 below) will check that the result satisfies the `makeList` specification.

Nested calls to simple fragments are allowed, so an alternative implementation of `makeList` is simply 'return `cons(e,nil)`'.

4.3 Recursion and variants

Recursive calls and mutual recursion are allowed — e.g. fragment `frag1` can call fragment `frag2`, where fragment `frag2` calls fragment `frag1` — provided the recursion eventually terminates. In order to prove termination, the CARE user supplies a variant function (or *variant* for short) whose value decreases on recursive calls.[4] For the purposes of this paper, a variant will be an $\mathbb{N}$-valued function defined on the input variables, although in the full CARE method, the variant consists of a measure and a well-founded ordering.

Here is a simple fragment which finds the last element in a linked list:

> Fragment `end(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `e:Element` such that $e = \mathit{last}(s)$.
> implementation:
>   assign `cdr(s)` to `t:List`;
>   case `null(t)` of
>   `yes:` return `car(s)`.
>   `no:` return `end(t)`.
> variant: $\#s$.

In executing this fragment on input `s`, the tail of `s` gets assigned to `t`, and then execution branches according to the result of `null(t)`. If `t` is empty (so `null(t)`

[4] With a mutually recursive set of fragments, it may not be necessary to give a variant for all of the fragments in the set: see [18] for details.

reports `yes`) then the value of `car(s)` is returned. Otherwise, the `end` fragment is called recursively on `t`. The length of the input list has been given as the variant.

4.4 Branching fragments

In addition to outputs (if any), implementations of branching fragments have reports at the leaves of the tree. Here is a branching fragment for "decomposing" a list:

> Branching fragment `decomposeList(s:List)` has
> specification:
>   result defined by cases:
>   if $\#s = 0$ then report `empty`
>   else report `nonempty`
>   with outputs `h:Element,t:List` such that $s = \langle h \rangle \frown t$.
> implementation:
>   case `null(s)` of
>   `yes:` report `empty`.
>   `no:` report `nonempty` and return `car(s),cdr(s)`.

Execution of `decomposeList(s)` has two possible cases. If `s` is empty, it reports `empty` without returning any outputs. Otherwise it reports `nonempty` and returns the head and tail of `s`.

4.5 Abort

An *abort* statement is provided for use in branches which will never be executed. For the purposes of illustration, here is a (rather artificial) example of a simple fragment which finds the second element in a list:

> Fragment `cadr(s:List)` has
> specification:
>   precondition $\#s \geq 2$
>   output `e:Element` such that $e = s(2)$.
> implementation:
>   case `decomposeList(s)` of
>   `empty:` abort.
>   `nonempty:` assign outputs to `a:Element,u:List`;
>   case `decomposeList(u)` of
>   `empty:` abort.
>   `nonempty:` assign outputs to `b:Element,v:List`;
>   return `b`.

In this example neither `empty` case will arise, since the precondition ensures that the input list has two or more elements. The abort statement signals to the proof obligation generator that it is necessary to show that the branch will never be executed.

4.6 Auxiliary fragments

As in functional programming, a complete program typically involves defining a set of "auxiliary" fragments in addition to the main fragment being defined. For example, here is a pair of fragments for reversing a list:

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 $w = rev(u) \frown v$.
implementation:
  case `decomposeList(u)` of
    `empty:` return `v`.
    `nonempty:` assign outputs to `h:Element,t:List`;
      return `revAcc(t,cons(h,v))`.
variant: $\#u$.

In effect, the auxiliary fragment `revAcc(u,v)` works its way along the list `u`, prepending each successive element onto the front of the "accumulator" `v`. In the fragment `reverse(s)`, the accumulator `v` is initialized to be the empty list `nil` and `u` is set equal to `s`. Thus, when the end of `u` is reached, `v` holds the reverse of the original value of `s`, as required.

5 Fragment verification

The purpose of fragment verification is to check that a fragment's implementation satisfies its specification. This section outlines an informal semantics for fragments and describes how to reason about their correctness.

5.1 Proof obligations

Under the CARE method, 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 I/O relationship.

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 discharged (i.e., 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 I/O relationship.

In practice, the proof obligations are generated 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 I/O relationship. The proof obligations are illustrated on examples below. The interested reader is referred to [18] for a more detailed treatment of proof obligation generation and its justification.

5.2 Partial correctness

For simple fragments, each (non-aborting) leaf must satisfy the I/O relationship given in the fragment's specification. The fragment's precondition can be assumed to hold. Consider for example the `reverse` fragment given in Section 4.6 above and repeated here for convenience.

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

Since both `nil` and `revAcc` have no precondition and the implementation is not recursive, there is only partial correctness to consider. In this case the tree has a single leaf, and the proof obligation is to show that the value r defined by `revAcc(s,nil)` satisfies $r = rev(s)$. From its specification, we know `nil` returns a list `t` such that $\#t = 0$, or in other words that t is the empty list $\langle\,\rangle$. From the specification of `revAcc`, we can assume that the list `r` returned by `revAcc(s,t)` satisfies $r = rev(s) \frown t$, and hence that $r = rev(s) \frown \langle\,\rangle = rev(s)$, as required.

For branching fragments, the situation is slightly more complicated because it is necessary to check that the report is appropriate and that the output satisfies the I/O relationship corresponding to the report. For example, consider the branching fragment example `decomposeList` from Section 4.4.

> Branching fragment `decomposeList(s:List)` has
> specification:
>   result defined by cases:

if $\#s = 0$ then report `empty`
else report `nonempty`
with outputs `h:Element,t:List` such that $s = \langle h \rangle \frown t$.
implementation:
case `null(s)` of
`yes`: report `empty`.
`no`: report `nonempty` and return `car(s),cdr(s)`.

The partial correctness proof obligations amount to checking that, if execution reaches a certain leaf then an appropriate result is returned. To reach the first leaf, for example, `null(s)` must have reported `yes` and thus $\#s = 0$, so 'report `empty`' is the appropriate result. To reach the second leaf, `null(s)` must have reported `no`, so $\#s \neq 0$ and '`nonempty`' is the appropriate report; from the specifications of `car` and `cdr`, the outputs `h` and `t` satisfy $h = \mathit{head}\, s$ and $t = \mathit{tail}\, s$, so $s = \langle h \rangle \frown t$ as required.

5.3 Termination

As a more complicated example, consider the `revAcc` fragment:

Fragment `revAcc(u:List,v:List)` has
specification:
output `w:List` such that $w = \mathit{rev}(u) \frown v$.
implementation:
case `decomposeList(u)` of
`empty`: return `v`.
`nonempty`: assign outputs to `h:Element,t:List`;
return `revAcc(t,cons(h,v))`.
variant: $\#u$.

For partial correctness, the I/O relationship which is required to hold at the leaves of the tree is that $w = \mathit{rev}(u) \frown v$. For this example the tree has two leaves. The first ('return `v`') arises when `decomposeList(u)` reports `empty`, or in other words when `u` is the empty list; in this case, the value of `v` is assigned to `w` and

$$\mathit{rev}(u) \frown v = \mathit{rev}\langle\,\rangle \frown w = \langle\,\rangle \frown w = w$$

as required. The second case arises when `decomposeList(u)` reports `nonempty`. In this case, the outputs of `decomposeList(u)` are assigned to `h` and `t`. From the specification of `decomposeList` we can assume that $u = \langle h \rangle \frown t$. The result `w` returned at the leaf is `revAcc(t,cons(h,v))` which, according to the specifications of `cons` and `revAcc` (used inductively), has value $\mathit{rev}(t) \frown (\langle h \rangle \frown v)$. It follows from simple properties of sequences that

$$\mathit{rev}(u) \frown v = \mathit{rev}(\langle h \rangle \frown t) \frown v = (\mathit{rev}(t) \frown \langle h \rangle) \frown v = \mathit{rev}(t) \frown (\langle h \rangle \frown v) = w$$

as required.

For termination, we need to show that the variant is decreasing on recursive calls. In this case, the value of the variant on the recursive call `revAcc(t,cons(h,v))` is $\#t$, and $\#u = \#(\langle h \rangle \frown t) = \#t + 1$, thus $\#t < \#u$ as required.

5.4 Well-formedness

To illustrate the well-formedness proof obligation consider the `end` example from Section 4.3.

> Fragment `end(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `e:Element` such that $e = last(s)$.
> implementation:
>   assign `cdr(s)` to `t:List`;
>   case `null(t)` of
>   `yes:` return `car(s)`.
>   `no:` return `end(t)`.
> variant: $\#s$.

In this implementation, the following fragment calls have nontrivial preconditions: `cdr(s)`, `car(s)`, `end(t)`. Well-formedness of the first two calls follows from the precondition of the fragment being verified (namely, $\#s \neq 0$). Well-formedness of `end(t)` requires establishing that $\#t \neq 0$, which follows from the fact that `null(t)` must have returned `no` in order for execution to have reached this point.

As an aside, partial correctness of `end` follows from the following two facts:

$$\#s \neq 0 \wedge \#\, tail(s) = 0 \Rightarrow last(s) = head(s)$$
$$\#s \neq 0 \wedge \#\, tail(s) \neq 0 \Rightarrow last(s) = last(tail(s))$$

5.5 Non-execution

Finally, as an example of the non-execution proof obligation, consider the second leaf in the implementation tree for `cadr(s)` in Section 4.5.

> Fragment `cadr(s:List)` has
> specification:
>   precondition $\#s \geq 2$
>   output `e:Element` such that $e = s(2)$.
> implementation:
>   case `decomposeList(s)` of
>   `empty:`   abort.
>   `nonempty:` assign outputs to `a:Element,u:List`;
>   case `decomposeList(u)` of
>   `empty:`   abort.
>   `nonempty:` assign outputs to `b:Element,v:List`;
>   return `b`.

From the fragment's precondition we can assume that $\#s \geq 2$, and from the specification of `decomposeList(s)` we can assume $s = \langle a \rangle ^\frown u$. To show that `decomposeList(u)` cannot report `empty` we need to show that $\#u \neq 0$, which follows from the above assumptions.

6 An example development

This section illustrates the use of the CARE method by giving a stepwise development of an algorithm for finding the integer part of the square root of a natural number [24]. The algorithm is developed through a series of design choices and each stage in the design is verified before passing to the next stage.

6.1 Program specification

The program has fragment specification

> Fragment `sqroot(s:Natnum)` has
> specification:
>   output `r:Natnum` such that $r^2 \leq s < (r+1)^2$.

6.2 First design step

The first step in a development of a program to satisfy this specification might be to introduce a new local variables `lo` and `hi` initialized to 0 and $s+1$ respectively, and then — keeping $lo^2 \leq s < hi^2$ invariant — to bring `lo` and `hi` progressively closer together until $hi = lo+1$. Such a design would be expressed in CARE notation as follows:

> Fragment `sqroot(s:Natnum)` has
> implementation:
>   assign `zero` to `lo:Natnum`;
>   assign `incr(s)` to `hi:Natnum`;
>   return `iterate(s,lo,hi)`.

where

> Fragment `zero` has
> specification:
>   output `n:Natnum` such that $n = 0$.

> Fragment `incr(m:Natnum)` has
> specification:
>   output `n:Natnum` such that $n = m + 1$.

are fragments one could expect to find in the library and

> Fragment `iterate(s,lo,hi:Natnum)` has
> specification:
>   precondition: $lo < hi \wedge lo^2 \leq s < hi^2$
>   output `r:Natnum` such that $r^2 \leq s < (r+1)^2$.
> implementation:

```
        case lessthan(incr(lo),hi) of
          yes: assign closeGap(s,lo,hi) to lo,hi:Natnum;
               return iterate(s,lo,hi).
          no:  return lo.
    variant: hi − lo.
```

is a new, looping fragment which performs the necessary iteration using auxiliary fragment `closeGap`, and

```
    Branching fragment lessthan(x,y:Natnum) has
    specification:
        result defined by cases:
            if x < y then report yes
            else report no.
```

would be another library fragment. `closeGap` is used to close the gap between `lo` and `hi`, and has the following specification:

> Fragment `closeGap(s,lo,hi:Natnum)` has
> specification:
> precondition $lo+1 < hi \wedge lo^2 \leq s < hi^2$
> output `u,v:Natnum` such that $u^2 \leq s < v^2 \wedge 0 \leq v - u < hi - lo$.

As well as preserving the invariant, `closeGap` ensures that the variant of `iterate` decreases on recursive calls. As we shall see, the form of the specification of `closeGap` is largely dictated by the proof obligations for `iterate`.

6.3 Verification of first design step

Having expressed the design, let us now verify it. There are two proof obligations associated with the `sqroot` fragment. Using the specifications of `zero` and `incr`, well-formedness of the call to `iterate` from `sqroot` involves showing

$$\forall s : \text{seq}\,\mathbb{N} \bullet 0 < s+1 \wedge 0^2 \leq s < (s+1)^2$$

Note that an error in the initialization of `lo` or `hi` (e.g. $hi = s$) would be revealed here. The partial correctness proof obligation for `sqroot` is to show that the result `r` returned by `iterate(s,lo,hi)` satisfies $r^2 \leq s < (r+1)^2$, but this follows immediately from the specification of `iterate`. This completes the verification of the `sqroot` fragment.

Turning next to `iterate`, partial correctness of the first leaf follows easily from the specifications of `closeGap` and `iterate` (used inductively). The partial correctness proof obligation for the second leaf amounts to showing

$$(lo < hi \wedge lo^2 \leq s < hi^2 \wedge lo+1 \not< hi) \Rightarrow lo^2 \leq s < (lo+1)^2$$

which follows from the fact that $lo < hi \wedge lo+1 \not< hi \Rightarrow hi = lo+1$. The termination proof obligation for `iterate` follows immediately from the specification of `closeGap`. The other proof obligations for `iterate` are straightforward.

Verifying the proof obligations for `iterate` gives us confidence that all of the salient information for `closeGap` has been captured in its specification.

6.4 Second design step

The next step in the development might be to refine `closeGap` by choosing a point mid somewhere between lo and hi and — by comparing the value of mid^2 with s — adjusting the value of lo or hi to equal mid appropriately. This could be expressed by implementing `closeGap` as follows:

Fragment `closeGap(s,lo,hi:Natnum)` has
implementation:
  assign `chooseIntermed(lo,hi)` to `mid:Natnum`;
  return `adjustBnds(s,lo,mid,hi)`.

where

Fragment `chooseIntermed(lo,hi:Natnum)` has
specification:
  precondition: $lo+1 < hi$
  output `mid:Natnum` such that $lo < mid < hi$.

and

Fragment `adjustBnds(s,lo,mid,hi:Natnum)` has
specification:
  precondition: $lo < mid < hi \land lo^2 \leq s < hi^2$
  output `u,v:Natnum` such that $u^2 \leq s < v^2 \land 0 \leq v - u < hi - lo$.
implementation:
  cases `lessthan(s,square(mid))` of
    `yes`:return `lo,mid`.
    `no`: return `mid,hi`.

where

Fragment `square(m:Natnum)` has
specification:
  output `n:Natnum` such that $n = m^2$.

The partial correctness proof obligations for `adjustBnds` are

$$(lo < mid < hi \land lo^2 \leq s < hi^2 \land s < mid^2)$$
$$\Rightarrow (lo^2 \leq s < mid^2 \land 0 \leq mid - lo < hi - lo)$$
$$(lo < mid < hi \land lo^2 \leq s < hi^2 \land s \not< mid^2)$$
$$\Rightarrow (mid^2 \leq s < hi^2 \land 0 \leq hi - mid < hi - lo)$$

The other proof obligations are easy to check.

```
Program sqroot(s:Nat)
var lo,hi:Nat

  procedure iterate(in s:Nat, var lo,hi:Nat) is
  var mid:Nat
  begin
        mid := div2(lo+hi);
        if s < mid*mid then lo,hi := lo,mid else lo,hi := mid,hi
  end iterate;

begin
     lo := 0; hi := s+1;
  iter: if lo+1<hi then iterate(s,lo,hi); goto iter else return lo
end sqroot
```

Figure 3: The `sqroot` algorithm "synthesized" from the design.

6.5 Third design step

The final step in the development is to choose a value for *mid* such that $lo < mid < hi$. Let us simply take the "midpoint" of *lo* and *hi*:

> Fragment `chooseIntermed(lo,hi:Natnum)` has
> implementation:
>   return `div2(add(lo,hi))`.

where

> Fragment `add(x,y:Natnum)` has
> specification:
>   output `z:Natnum` such that $z = x + y$.

> Fragment `div2(m:Natnum)` has
> specification:
>   output `n:Natnum` such that $n = m$ div 2.

This completes the development of `sqroot`. Fig. 3 shows the algorithm that might be synthesized from this design. (The actual code synthesized by Care tools would obviously depend on what target language was used.)

7 An example programming technique

7.1 Templates

A *template* is a parameterized collection of CARE types and fragments, some fully implemented and others simply specified. One or more of the fully implemented fragments act as the *seed* of the template. A template may have *applicability conditions* which dictate the circumstances under which the template can be used.

To use a template, the CARE user instantiates the parameters in such a way that the template's seed matches one of the problems at hand (i.e., a fragment requiring implementation) and such that the applicability conditions can be discharged. The instantiated template gets added to the fragment set: the fully implemented fragments of the template become new auxiliary fragments, and the specified-only fragments become the new problems.

Templates can be written, for example, for choosing the data structures to implement abstract data types, or for defining algorithms to calculate required properties or to achieve desired ends. An example is given below which illustrates the use of a template to define a strategy for processing lists using an accumulator.

As indicated in Section 1, we envisage that common program refinement techniques would be expressed as reusable templates, whose correctness is established once, off-line, by a CARE expert. In this way, the software engineer's verification task would be reduced from proving the whole fragment set to showing that the template's applicability conditions are satisfied, which is generally a much simpler task.

7.2 An accumulator

A common strategy for defining programs which successively process the elements of a list is to use an *accumulator*, which is a variable that holds intermediate values as the list is processed. The strategy involves introducing a secondary procedure which takes the accumulating value as a parameter.

For example, let *sum* be the function which sums the elements of a list of numbers:

$$sum : \operatorname{seq} \mathbb{Z} \rightarrow \mathbb{Z}$$

$$sum\langle\, \rangle = 0$$
$$\forall\, x : \mathbb{Z} \bullet sum\langle x \rangle = x$$
$$\forall\, s, t : \operatorname{seq} \mathbb{Z} \bullet sum(s \frown t) = sum(s) + sum(t)$$

Here is a CARE program for calculating *sum*:

Fragment `sum(s:NatList)` has
specification:
  output `n:Natnum` such that $n = sum(s)$.
implementation:
  return `sumAcc(s,zero)`.

```
Fragment sumAcc(s:NatList,m:Natnum) has
specification:
    output n:Natnum such that n = sum(s) + m.
implementation:
    case decomposeList(s) of
      empty:     return m.
      nonempty: assign outputs to h:Natnum,t:NatList;
                 return sumAcc(t,add(m,h)).
variant: #s.
```

The result is a tail-recursive procedure whose algorithmic complexity is linear in the length of the list.

7.3 A template for accumulators

Generalizing from the above, a simple template for list accumulators is:

```
Fragment processList(s:ElemList) has
specification:
    output b:Acc such that b = f(s).
implementation:
    return accumulator(s,base).

Fragment accumulator(s:ElemList,a:Acc) has
specification:
    output b:Acc such that b = foldl(hd, a, s).
implementation:
    case decomposeElemList(s) of
      empty:     return a.
      nonempty: assign outputs to h:Elem,t:ElemList;
                 return accumulator(t,processElem(a,h)).
variant: #s.

Fragment processElem(b:Acc,x:Elem) has
specification:
    output c:Acc such that c = hd(b, x).

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

Branching fragment decomposeElemList(s:ElemList) has
specification:
    result defined by cases:
        if #s = 0 then report empty
        else report nonempty
            with outputs h:Elem,t:ElemList such that s = ⟨h⟩ ⌢ t.
```

where Acc, Elem and ElemList are CARE types with carrier sets '*Acc*', '*Elem*' and 'seq *Elem*' respectively, and *foldl* is the function for "folding left" [7] over a list:

$$
\begin{array}{l}
[X, Y] \\
\hline
foldl : (X \times Y \rightarrow X) \times X \times \mathrm{seq}\, Y \rightarrow X \\
\hline
\forall f : X \times Y \rightarrow X;\ x : X \bullet foldl(f, x, \langle\rangle) = x \\
\forall f : X \times Y \rightarrow X;\ x : X;\ h : Y;\ t : \mathrm{seq}\, Y \bullet \\
\quad foldl(f, x, \langle h \rangle \frown t) = foldl(f, f(x, h), t)
\end{array}
$$

The template's seed is the processList fragment. The non-type parameters of the template are given in Table 4. (*dh* is an auxiliary function — to be supplied by the user — which is used in the applicability conditions. Its purpose is to provide information which is required in discharging the template's proof obligations.) The names of the CARE types and fragments can be changed to suit the problem at hand.

Name	Signature	Explanation
f	$\mathrm{seq}\, Elem \rightarrow Acc$	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

Table 4: Value parameters of the template.

The applicability conditions for the template are:

1. $f\ \langle\rangle = base$
2. $\forall\, h : Elem;\ t : \mathrm{seq}\, Elem \bullet f(\langle h \rangle \frown t) = dh(h, f(t))$
3. $\forall\, x : Elem \bullet dh(x, base) = hd(base, x)$
4. $\forall\, x, y : Elem;\ a : Acc \bullet dh(x, hd(a, y)) = hd(dh(x, a), y)$

Note that when $Acc = Elem$ and hd is associative and commutative, it is sufficient to define $dh(x, a) \triangleq hd(a, x)$ and to establish the first two conditions.[5]

See Section 7.7 below for a proof that these conditions are sufficient to establish the correctness of the template. The proof involves induction and advanced properties of *foldl*, but by presenting the applicability conditions in the above form we have shielded the CARE user from such details. Verification of the applicability conditions requires only knowledge of the problem domain, as the examples below show.

Keith Harwood has developed a set of accumulator templates with a whole range of applicability conditions [16].

[5] A function $g : X \times X \rightarrow X$ is said to be associative and commutative if

$$
\begin{array}{l}
\forall\, x, y : X \bullet g(x, y) = g(y, x) \\
\forall\, x, y, z : X \bullet g(x, g(y, z)) = g(g(x, y), z)
\end{array}
$$

7.4 Example: summing a list

To implement the `sum` fragment using the accumulator template, instantiate the parameters as follows:

$$\begin{array}{rclrcl} Elem & \leadsto & \mathbb{N} & base & \leadsto & 0 \\ Acc & \leadsto & \mathbb{N} & hd(a,x) & \leadsto & a+x \\ f(s) & \leadsto & sum(s) & dh(x,a) & \leadsto & x+a \end{array}$$

The `Acc`, `Elem` and `ElemList` types could be implemented by `Natnum`, `Natnum` and `NatList` respectively. The `processElem`, `base` and `decomposeElemList` fragments could be implemented by `add`, `zero` and `decomposeList` from above, respectively. The resulting implementation of `sum` is then exactly as given in Section 7.2 above.

The applicability conditions become:

1. $sum\ \langle\,\rangle = 0$
2. $\forall\, h : \mathbb{N};\ t : \mathrm{seq}\,\mathbb{N} \bullet sum(\langle h \rangle \frown t) = sum(t) + h$
3. $\forall\, x : \mathbb{N} \bullet x + 0 = 0 + x$
4. $\forall\, x, y, z : \mathbb{N} \bullet x + (y + z) = (x + y) + z$

These are all valid statements, as is easily checked.

7.5 Example: reverse a list

To implement the `reverse` fragment

> Fragment `reverse(s:List)` has
> specification:
> output `r:List` such that $r = rev(s)$.

using the accumulator template, instantiate the parameters as follows:

$$\begin{array}{rclrcl} Elem & \leadsto & ELEMENTS & base & \leadsto & \langle\,\rangle \\ Acc & \leadsto & \mathrm{seq}\,ELEMENTS & hd(a,x) & \leadsto & \langle x \rangle \frown a \\ f(s) & \leadsto & rev(s) & dh(x,a) & \leadsto & a \frown \langle x \rangle \end{array}$$

The `Acc`, `Elem` and `ElemList` types could be implemented by `List`, `Element` and `List` respectively. The `base` and `decomposeElemList` fragments could be implemented by `nil` and `decomposeList` from above, respectively. The `processElem` fragment could be implemented as follows:

> Fragment `processElem(b:List,x:Element)` has
> specification:
> output `c:List` such that $c = b \frown \langle x \rangle$.
> implementation:
> `cons(x,b)`.

The resulting implementation of `reverse` is then essentially the same as given in Section 4.6 above.

The applicability conditions become:

1. $rev\langle\,\rangle = \langle\,\rangle$
2. $\forall\, h : ELEMENTS;\ t : \mathrm{seq}\, ELEMENTS \bullet rev(\langle h\rangle \frown t) = rev(t) \frown \langle h\rangle$
3. $\forall\, x : ELEMENTS \bullet \langle\,\rangle \frown \langle x\rangle = \langle x\rangle \frown \langle\,\rangle$
4. $\forall\, x, y : ELEMENTS;\ a : \mathrm{seq}\, ELEMENTS \bullet (\langle y\rangle \frown a) \frown \langle x\rangle = \langle y\rangle \frown (a \frown \langle x\rangle)$

These are all valid statements, as is easily checked.

7.6 Example: list maximum

Suppose `maximum` is a fragment for finding the maximum element of a list of numbers with the following specification:

> Fragment `maximum(s:NatList)` has
> specification:
> output `n:Natnum` such that $n = max(ran(s) \cup \{0\})$.

This can be implemented using an accumulator by instantiating the template's parameters as follows:

$$\begin{array}{rcl@{\qquad}rcl}
Elem & \leadsto & \mathbb{N} & base & \leadsto & 0 \\
Acc & \leadsto & \mathbb{N} & hd(a,x) & \leadsto & largerOf(a,x) \\
f(s) & \leadsto & max(ran(s) \cup \{0\}) & dh(x,a) & \leadsto & largerOf(a,x)
\end{array}$$

where

$$\begin{array}{|l}
largerOf : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \\
\hline
\forall\, x, y : \mathbb{Z} \bullet \text{if } x \leq y \\
\qquad\qquad \text{then } largerOf(x,y) = y \\
\qquad\qquad \text{else } largerOf(x,y) = x
\end{array}$$

The applicability conditions are easily checked.

7.7 Verification of the template

This section shows that the applicability conditions given in Section 7.3 are sufficient to establish the correctness of the `processList` and `accumulator` fragments.

Partial correctness of `accumulator` follows from the following facts:

$$foldl(hd, a, \langle\,\rangle) = a$$
$$foldl(hd, a, \langle h\rangle \frown t) = foldl(hd, hd(a,h), t)$$

Termination of `accumulator` follows from the fact that $\#t < \#s$.

The partial correctness proof obligation for `processList` is

$$\forall s : \mathrm{seq}\, Elem \bullet f(s) = foldl(hd, base, s)$$

which can be established by (left) induction on s as follows:

Base case

The base case is

$$f\ \langle\, \rangle = foldl(hd, base, \langle\, \rangle)$$

which follows immediately from condition (1) and the definition of *foldl*.

A useful lemma

For the induction step we first prove the following lemma:

$$\forall h : Elem;\ t : \mathrm{seq}\, Elem \bullet dh(h, foldl(hd, base, t)) = foldl(hd, base, \langle h\rangle \frown t)$$

by (right) induction on t. The base case is straightforward:

$$\begin{array}{lr}
dh(h, foldl(hd, base, \langle\, \rangle)) & \\
= dh(h, base) & \text{by properties of } foldl \\
= hd(base, h) & \text{by (3)} \\
= foldl(hd, base, \langle h\rangle) & \text{by properties of } foldl \\
= foldl(hd, base, \langle h\rangle \frown \langle\, \rangle) & \text{as required}
\end{array}$$

The induction step — from $t = u$ to $t = u \frown \langle e\rangle$ — is

$$\begin{array}{lr}
dh(h, foldl(hd, base, u \frown \langle e\rangle)) & \\
= dh(h, hd(e, foldl(hd, base, u))) & \text{by properties of } foldl \\
= hd(dh(h, foldl(hd, base, u)), e) & \text{by (4)} \\
= hd(foldl(hd, base, \langle h\rangle \frown u)), e) & \text{by the sub-induction hypothesis} \\
= foldl(hd, base, (\langle h\rangle \frown u) \frown \langle e\rangle) & \text{by properties of } foldl \\
= foldl(hd, base, \langle h\rangle \frown (u \frown \langle e\rangle)) & \text{as required}
\end{array}$$

The induction step

The induction step for the main proof amounts to proving

$$f(\langle h\rangle \frown t) = foldl(hd, base, \langle h\rangle \frown t)$$

from the induction hypothesis $f(t) = foldl(hd, base, t)$:

$$\begin{array}{lr}
f(\langle h\rangle \frown t) & \\
= dh(h, f(t)) & \text{by (2)} \\
= dh(h, foldl(hd, base, t)) & \text{by induction hypothesis} \\
= foldl(hd, base, \langle h\rangle \frown t) & \text{by the above lemma}
\end{array}$$

This completes the proof of the partial correctness of `processList`, and hence of the template.

8 Conclusions

This paper has outlined the Care method of development of formally verified software, concentrating on the development of algorithms from formal program specifications. The method structures the development process in such a way that engineering aspects are separated from formal mathematical aspects as far as possible. This is achieved by factoring the verification task into several parts:

- generalized design steps (templates) are verified off-line;
- problem-specific design steps are accomplished by instantiating templates and verifying applicability conditions;
- the interface to the target language is restricted to primitive fragments and types (corresponding roughly to standard library routines), which are verified off-line by appropriate means.

The result is a framework which supports reuse of design fragments and hiding of verification detail. Under the Care approach, software development becomes a process of selection and instantiation of fragments, and verification amounts to checking "correctness of fit" of the fragments, using a formal requirements specification as a guide. As far as possible, the goal has been to consign formal verification aspects of the method to automated tools, allowing the software engineer to concentrate instead on the design and development of usable, efficient pieces of software.

The method is general and can be used in conjunction with a variety of other methods, both formal and informal. It can be used with a wide variety of specification languages, theorem provers and target languages.

A series of prototype tools have been built to support the method, including parsers, syntax/type-checkers, pretty-printers, proof obligation generators, mathematical simplifiers, an automatic theorem prover, automated supported for formal reasoning in an interactive theorem prover, and a code synthesizer with C as target language. A large library of pre-proven fragment templates has been produced, together with a tool which assists software engineers in selecting and instantiating templates.

Acknowledgements

The work reported in this paper is part of a collaborative project involving the following people: Keith Harwood, Thies Arens, Frances Collis, David Hemer, Rex Matthews and Trudy Weibel. The general applicability conditions for the accumulator were discovered by Keith, and Trudy supplied the original proof of sufficiency. The author would also like to thank a number of his colleagues at the SVRC for their useful contributions, including John Staples and Ian Hayes.

References

[1] Trusted Computer System Evaluation Criteria. U.S. Dept of Defense, December 1985. Standard 5200.28-STD (Orange Book).

[2] Information Technology Security Evaluation Criteria (ITSEC). Commission of the European Communities, June 1991. Provisional Harmonised Criteria.

[3] Safety related software for railway signalling. U.K. Railway Industry Association, 1991. Technical specification no 23.

[4] The Procurement of Safety Critical Software in Defence Equipment. U.K. Ministry of Defence, April 1991. Interim Defence Standard 00-55.

[5] Assessment of munition related safety critical computing systems. Australian Ordnance Council, August 1993. Pillar Proceeding 223.93.

[6] S. Austin and G. Parkin. Formal methods: A survey. Technical report, National Physical Laboratory, Dept of Trade and Industry, Middlesex, United Kindgom, March 1993.

[7] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[8] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[9] S.M. Brien and J.E. Nicholls. Z Base Standard, Version 1.0. Technical Report SRC D–132, Oxford University Programming Research Group, November 1992.

[10] J. Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[11] D. Craigen et al. Eves: an overview. In *Proceedings of VDM'91.* Springer-Verlag, 1991.

[12] D. Good. Mechanical proofs about computer programs. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 55–75. Prentice Hall International, 1985.

[13] J.V. Guttag and J.J. Horning. Report on the larch shared language. *Science of Computer Programming*, 6:103–134, 1986.

[14] K. Harwood, P. Lindsay, and R. Matthews. An Approach to Constructing Verified Software. In *Seventeenth Annual Computer Science Conference*, pages 777–786, University of Canterbury, Christchurch, January 1994.

[15] Keith Harwood. Towards tools for formal correctness. In *The Fifth Australian Software Engineering Conference*, pages 153–158. The Institution of Radio and Electronics Engineers Australia, May 1990.

[16] Keith Harwood. The accumulator fragment. Technical Report DCS 14664-01, Telectronics Pacing Systems, Sydney, R&D, Aug. 1994.

[17] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEE/BCS Software Engineering Journal*, 4(6):330–338, November 1989.

[18] D. Hemer and P.A. Lindsay. Formal specification of proof obligation generation in CARE. Technical Report 95-13, Software Verification Research Centre, University of Queensland, 1995.

[19] David Hemer, Peter Lindsay, and Rex Matthews. Formal specification of an abstract syntax for fragments. Technical Report 93-9, Software Verification Research Centre, The University of Queensland, Australia, Nov. 1993. Version 2.5.

[20] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, second edition, 1990.

[21] P. Lindsay. Expressing program developments from the refinement calculus in care. Technical Report 94-6, Software Verification Research Centre, University of Queensland, 1994.

[22] P.A. Lindsay. The data logger case study in CARE. Technical Report 95-10, Software Verification Research Centre, University of Queensland, 1995.

[23] Rex Matthews and Trudy Weibel. Code synthesis in CARE. Technical report, Software Verification Research Centre, The University of Queensland, Australia, Sep. 1994. working paper.

[24] C. Morgan. *Programming from Specifications.* Prentice-Hall International, 1990.

[25] M. Naftalin, T. Denvir, and M. Bertran, editors. *2nd International Symposium of Formal Methods Europe.* Springer Verlag, October 1994. Lecture Notes in Comp. Sci. Vol. 873.

[26] S. Sokolowski. Partial correctness: the term-wise approach. *Science of Computer Programming*, 4:141–157, 1984.

[27] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[28] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Proceedings 12th International Conference on Automated Deduction*, pages 341–355, June 1994.

[29] Mark Utting and Keith Whitwell. Ergo user manual. Technical Report 93-19, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, March 1994.

[30] J. Woodcock and P.G. Larsen, editors. *1st International Symposium of Formal Methods Europe.* Springer Verlag, April 1993. Lecture Notes in Comp. Sci. Vol. 670.