SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-52

The CARE toolset for developing verified programs from formal specifications

David Hemer and Peter Lindsay

December 1995

Phone: +61 7 3365 1003
Fax: +61 7 3365 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 toolset for developing verified programs from formal specifications

David Hemer and Peter Lindsay
Software Verification Research Centre
The University of Queensland, St Lucia Qld 4072

15 December 1995

Abstract

This paper describes the CARE toolset for interactive development of verified programs from formal specifications. The software engineer begins by giving a characterization of the application domain in the form of a mathematical theory. CARE tools are then used to progressively design a program by sketching out the program structure and gradually filling in the details. At any stage the correctness of the partial design can be checked by using one of the CARE tools to generate proof obligations. Another tool gives access to pre-proven parameterised design templates which encapsulate useful programming knowledge. When the design is complete, a third CARE tool is used to automatically synthesize a source code program which – if all the proof obligations can be discharged – is guaranteed to meet its formal specification.

The knowledge base of CARE can be extended by users in a soundness-preserving manner to include reusable domain theories, library routines, design templates and proof tactics. The CARE toolset includes a fully automatic resolution-based theorem prover which will discharge many of the simpler proof obligations, and a general-purpose interactive theorem prover for the rest.

1 Introduction

1.1 Motivation

The need for formal methods during the software development is becoming increasingly recognized by the software engineering community, particularly in critical applications [13]. This is evidenced by the emergence of a number of software standards that insist formal methods be used at various stages of the software development lifecycle, for example for security critical [4] and safety-critical applications [5]. Many companies, especially in Europe, are now using formal specification techniques but the next step – formal verification of programs – is generally regarded as a difficult, time-consuming task requiring esoteric mathematical skills.

The CARE project aims to bring formal verification within the reach of software engineers trained in formal specification. The CARE system addresses the problem in two ways: firstly, by providing a framework within which programming knowledge can be recorded and reused

*The CARE project is a collaboration between the SVRC and Telecommunications 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.
with minimal need for re-proof; and secondly by providing tools for generating and discharging conditions which guarantee that the program meets its specification. We feel that the combination of interactive tools with appropriate libraries of reusable, formally verified programming knowledge is the best route to realizing the aim of making formal verification feasible for software engineers with an understanding of formal specification.

A major innovation of the CARE approach is that it supports the development of verified software for target languages which themselves do not have a formal semantics. It does this by restricting the user's use of target-language code to formally specified library routines which are verified "off-line". Code in the target language is automatically synthesized from programs written in the CARE language.

1.2 Program development using CARE

CARE stands for Computer Assisted Refinement Engineering. The CARE prototype toolset supports a simplified version of the well-established VDM approach to program development from formal specifications [6, 8, 11, 17]. 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. 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 characterizes the problem domain within which the application is to be developed. The theory typically consists of mathematical definitions of the types of objects 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 [10, 29]. 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 [14].

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. (The prototype toolset has C as its target language, but the examples in this paper will use Pascal, for clarity.) 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.

The CARE toolset includes a fully automatic resolution-based theorem prover which will discharge many of the simpler proof obligations, and a general-purpose interactive theorem prover for the more difficult ones. We have found that many of the steps involved in going from high-level abstract specifications to efficient executable implementations depend on leaps of mathematical insight which seem to be well beyond the abilities of current generations of automatic theorem provers [22]. In interactive theorem provers such as HOL [1] and Isabelle [2], however, such leaps can be encoded as proof tactics. The CARE library will be populated with general and domain-specific proof tactics, with the intention that CARE users experiment with different combinations of tactics until they find proofs (or refutations) of their proof obligations.

**Code synthesis:** When the program design is complete, another CARE tool is used to automatically synthesize 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 synthesized 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 characterize 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.

The CARE toolset comes supplied with a library of pre-proven, parameterised design "templates" which the user can instantiate to suit the problem at hand. The knowledge base of CARE can be extended by users in a soundness-preserving manner to include reusable domain theories, design strategies, primitive components, proof tactics, and so on.

The prototype toolset has been populated with a large number of design templates and primitive components for numbers, lists, arrays and records. We have used the CARE method on a number of medium-sized applications including verification of the design of an event logger such as might be used in an embedded device [24]. Work is proceeding on populating the tools with further general purpose design templates and primitive components and developing larger case studies. Section 4 illustrates the use of the CARE method on the 'spellchecker' problem.

### 1.3 The CARE project

The CARE project is a collaboration between Telectronics Pacing Systems and the Software Verification Research Centre. Telectronics develops and manufactures software-driven medical devices such as implantable defibrillators. The company has long been motivated to investigate the use of formal methods for the economical and timely development of prov-
ably correct software. A grant from the Australian Government has enabled more extensive
development of the ideas and the construction of a prototype toolset to support the method.
The CARE method has been trialled at Telecommunications in a five-day intensive training course
attended by senior software engineers not directly involved in the CARE project. It is testi-
mony to the effectiveness of the method and toolset that it can be used after so little training
(albeit for fairly small examples). Telecommunications have now decided to try the approach to
develop part of the software for their next product range.

1.4 This paper

Section 2 describes the CARE language for the development and verification of programs.
Section 3 outlines the functionality of the tools in the CARE toolset. Section 4 illustrates the
use of CARE to develop a small application. Section 5 compares the CARE approach with
other systems for formal verification.

2 Brief description of CARE

This section briefly describes the CARE notation: the interested reader is referred to one of
the CARE overview documents for more details [23, 25].

2.1 Theories

Theories allow the software engineer to define a mathematical representation of the “domain
theory” of the application: that is, the concepts, objects and relationships which characterize
the desired application program. (The CARE toolset accepts Z-like ascii notation for mathe-
matical expressions and pretty-prints them in their more familiar form.) Each new constant,
function and relation is defined via a signature and a set of axioms. For example, here are
theory components that define functions append for appending an element onto a sequence
and elements for finding the set of elements of a sequence:

Theory Definition of append.
append : E × seq E → seq E;
∀ x : E, s : seq E • append(x, s) = (x) ⊔ s.

Theory Definition of elements
elements : seq E → ℘ E;
elements(Æ) = Ø,
∀ x : E • elements(Æ) = {x},
∀ s, t : seq E • elements(s ⊔ t) = elements(s) ⊔ elements(t).

The library supplied with the CARE prototype toolset contains definitions of many of the
constructs from the Z mathematical toolkit.

2.2 Program components

CARE programs have two different kinds of components: types for expressing data structures
and fragments for expressing algorithms. In a complete CARE program every component
has a specification (mathematical characterization) and an implementation (from which code
will eventually be synthesized). The implementation can be either directly in code or it can
be defined in terms of other components; the former are called primitive components and
the latter higher-level components. The implementation of primitive components, and the
proof that the associated target language code satisfies the corresponding specification, are
c onsidered to be outside the scope of CARE.
For clarity, the names of CARE components and variables are written in typewriter font,
and mathematical expressions are written in italics. In what follows we use Pascal as our
target language, for didactic purposes.

2.3 Types

Each object within a CARE program has an associated type, which describes the kind of
values that the object may take. Types are specified by giving a mathematical expression
representing the set of values that an object of this type may take. For example, here are
specifications of types for lists and sets:

Type List has specification: seq X
Type Set has specification: \( \exists X \)

Higher-level types are implemented in terms of other types by defining a data refinement.
(See e.g. [17] for an explanation of data refinement.) For example, sets can be implemented
as non-repeating sequences:

Type Set has implementation:
value s:Set is refined by x:List
  with invariant \( \forall i,j:1..\#x \cdot i \neq j \Rightarrow x(i) \neq x(j) \)
  with refinement relation \( s = elements(x) \).

Under such a refinement, a set can be represented by any non-repeating list which has the
same elements. The invariant says that only non-repeating lists will be used to represent
sets; the refinement relation describes how to view a non-repeating list as a set. When a data
refinement has been chosen, the software engineer can then proceed to implement operations
on object of the abstract type in terms of operations on the corresponding objects of the
more concrete type(s); e.g. finding the cardinality of a set simply becomes finding the length
of a corresponding list. The refinement relation and invariant play a role in the associated
verification: see [24] for examples.

2.4 Fragments

There are two kinds of fragments: simple and branching.

Simple fragments are roughly analogous to functions in a procedural programming language.
The specification of a simple fragment defines the types of its inputs and outputs, the frag-
ment’s precondition, and the relationship between the inputs and outputs. The precondition
is a constraint on the possible inputs to the fragment; it is specified as a predicate of
the input values. In many cases the precondition is simply ‘true’, in which case it will be
omitted. There are proof obligations to check that the precondition is satisfied each time the
fragment is called.

As an example, here is the specification of a simple fragment named first which takes a
single input \( x \) of type List such that \( x \) is non-empty; it returns a single output \( e \) of type
Element such that \( e \) is the first element of \( x \).

Fragment first(x:List) has
specification:
  precondition \( x \neq () \)
  output \( e:Element \) such that \( e = head x \)
The main difference between simple and branching fragments is that the latter are used for branching of control. Each branch is identified by a label called its report. The flow of control is determined by tests called guards, which are specified as predicates of the input values. Conceptually, each of the guards is evaluated in turn until one is found that is true; by default no guard is given for the last branch, so it is taken if all the preceding guards are false. The number and type of outputs returned by a branching fragment may be different for different branches, and some branches may return no result at all.

For example, here is the specification of a branching fragment decomposeList which takes a single argument x of type List and tries to decompose it. If x is the empty list, then decomposeList(x) simply reports empty with no outputs. Otherwise, it reports nonempty and returns two values h and t such that x is equal to the list formed by appending h onto the front of t (in other words, h is the head of x and t is the tail).

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

2.5 Fragment implementations

Primitive fragments are implemented directly as target language code: simple fragments are defined as target language functions; branching fragments are defined by giving Boolean-valued tests for the guards and function definitions for each branch. See Section 4.6 for examples.

The implementation of a higher-level fragment is tree-structured, with fragment calls at the nodes. Non-branching nodes of the tree correspond to bindings of values to local variables; the values of the bindings are either variables or the result of a simple fragment call. Branching nodes correspond to calls to branching fragments, and in the case where a branch returns values, these are bound to local variables on the appropriate branch. Each leaf node of the tree contains the result returned (if any), together with a report in the case of branching fragments.

To illustrate, let us suppose that first and rest are simple fragments that return the head and tail of a non-empty list. (first was specified above.) Suppose also that isEmpty is a branching fragment which takes a list and simply reports yes or no, depending on whether or not the list is empty. Using these fragments, here is an implementation for the fragment decomposeList which was specified earlier:

Branching fragment decomposeList(x:List) has
implementation:
  cases isEmpty(x) of:
     yes: report empty
     no: assign first(x) to h:Element;
         assign rest(x) to t:List;
         report nonempty and return h, t

In this case, the root of the implementation tree contains a call to isEmpty. If the list x is empty, control passes to the branch labelled yes, and empty is reported with no value returned. Otherwise, control passes to the branch labelled no, where the result of the fragment call first(x) is bound to local variable h, then the result of fragment call rest is bound to local variable t, and the values of h and t are returned, with report nonempty.
Fragment implementations may involve recursive calls, including mutual recursion. In order to establish termination of fragment evaluation, the user should supply a variant function (that is, an N-valued function of the input variables) whose value decreases on recursive calls: see [15] for full details.

2.6 The library and templates

The CARE library is a repository for a number of pre-proven, reusable packages called templates. In its most general form, a template consists of a collection of fragments, types and theories, which together implement some algorithm or data structure.

Templates may be parameterised, and a given template can be instantiated in a number of different ways to suit different problems. There may be constraints on the formal parameters, called applicability conditions, which express the properties which enable the template to be verified in its most general form. When the template is instantiated, the corresponding instances of the applicability conditions become proof obligations for the implementations introduced by that template. This means that many of the difficult modelling and verification steps can be done “off-line” by verification experts and then packaged for use by the software engineer, who is left with the considerably simpler task of verifying the applicability conditions. Section 4 below gives some examples.

2.7 Proof obligations

Proof obligations are the conditions which ensure that implementations of higher-level components satisfy their specifications. For example, assuming that first is as specified as above, that rest(x) has specification tail x, and that the guard of isEmpty(x) is x = {}, then here is the proof obligation which checks the appropriateness of the report and outputs at the second leaf in the implementation of decomposeList above:

\[
\begin{align*}
\text{Proof obligation decomposeList\_pc\_path2} & \equiv \forall x : \text{seq} X \bullet (x = \{\}) \Rightarrow \\
& \forall h : X \bullet h = \text{head}(x) \Rightarrow \\
& \forall t : \text{seq} X \bullet t = \text{tail}(x) \Rightarrow \\
& \neg (x = \{\}) \land x = \text{append}(h, t)
\end{align*}
\]

In words, the proof obligation says that, if x is non-empty, h is the head of x and t is the tail of x, then nonempty is the appropriate report and output pair (h, t) satisfies the corresponding input/output relationship.

In brief, the different kinds of proof obligation are:

Partial correctness: Each leaf node in an implementation tree satisfies the required input/output relationship.

Well-formedness: The precondition of each fragment call is satisfied.

Applicability conditions: Templates’ parameters have been instantiated in a valid manner.

Termination: Variant functions are supplied for all recursive fragments, and the value of the variant decreases for each recursive call.

The interested reader is referred to [15] for full details of the proof obligation generation process.
3 The CARE toolset

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 synthesizer, were themselves formally specified in Z (see e.g. [15]) and implemented by an almost mechanical translation to Prolog. The overall architecture of the toolset is shown in Fig. 1.

![Architecture of the CARE toolset](image)

**Figure 1: 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”. 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 be added to the worksheet or modified only via the script.

Library: The library consists of a collection of pre-proven design templates. Each template contains any number of the following: formal parameters and applicability conditions; theory declarations, including constant, function and relation signatures, theorems and lemmas; fragment and type specifications, with or without implementations; proof tactics; documentation. There is a tool to help the user search the library.

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. It incorporates a parser for declarations in Ascii, together with tools for retrieving and instantiating templates from the library (see below).

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. There are directives for renaming or omitting nominated components of the template. The template instantiator calculates a “minimal closure” of the components that will be needed to make a self-contained component set.

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. Both theorem provers are more or less stand-alone tools. Keith Harwood and his team at Teletronics have developed a purpose-built automatic theorem prover for CARE based on order-sorted resolution under equality [32]. The SVRC team has adapted the Ergo interactive proof assistant [31] by making use of its store of theorems about many-sorted set theory and developing special-purpose simplification procedures and appropriate tactics and heuristics for CARE proof obligations.

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. (For example, the status of a proof obligation is determined directly from the output of the automatic theorem prover, or by polling the status of the corresponding conjecture in the interactive theorem prover upon request via a script command.) 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.

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). There is also a pretty printer to display the syntax trees as Prolog terms, for debugging purposes.

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 optimizations along the way. The reader is referred to [26] for further details on the code synthesis process. The current prototype supports tail recursion but we plan to extend this to more general forms of recursion in the near future.

4 Example development of a spellchecker program

This section outlines the development of a simple application using the CARE toolset.

4.1 A top-level design for a spellchecker

The problem we shall consider is the design of a system which reports misspelt words in a file. This problem can be split into the following subproblems:

1. parse the file to get a list of words;
2. sort the list of words into a lexicographically ordered list;
3. compare the sorted list with a dictionary, stepping through the two lists in parallel and storing the misspelt words;
4. output the remaining words to the user who may then have the opportunity to either correct the misspelt word, or add it as a new word to the dictionary.

Assuming the number of different words in a file is generally much smaller than the total number of words in the dictionary, this design will be fairly efficient since it makes a single pass (only) through the dictionary. We shall concentrate here on the second and third stages of the design.

4.2 Domain theory

This section outlines a domain theory for the above design. First, we introduce a new type $Word$ to represent the set of all possible words. Let $before$ be a binary relation representing the lexicographic ordering on words. Rather than giving a full definition at this stage we simply state some useful properties of $before$: namely that any two words can be compared, and that the relationship is transitive.

Theory Definition of $before$

$before : Word \times Word$;
$\forall x, y : Word \bullet before(x, y) \lor x = y \lor before(y, x),$  
$\forall x, y, z : Word \bullet before(x, y) \land before(y, z) \Rightarrow before(x, z).$

The predicate $ordered$ represents lexicographically ordered word sequences.

Theory Definition of $ordered$

$ordered : seq Word;$
$ordered(\langle \rangle),$
$\forall h : Word, t : seq Word \bullet ordered(append(h, t)) \Leftrightarrow t = \langle \rangle \lor (before(h, head t) \land ordered(t))$

The dictionary is modelled as an ordered sequence of words:
Theory Definition of dictionary
\[ \text{dictionary} : () \rightarrow \text{seq} \text{ Word}; \]
\[ \text{ordered} \text{(dictionary)}. \]

The function \textit{remove} takes two lists and returns the set of words which are in the first list but not in the second:

Theory Definition of \textit{remove}
\[ \text{remove} : \text{seq} \text{ Word} \times \text{seq} \text{ Word} \rightarrow \text{F} \text{ Word}; \]
\[ \forall s, t : \text{seq} \text{ Word} \bullet \text{remove}(s, t) = \text{elements}(s) \setminus \text{elements}(t). \]

\textit{(elements} is defined in Section 2.1 above, and \( \setminus \) is the usual set difference operator.\textit{)}

4.3 Initial specification

Having defined an initial domain theory, we are now ready to specify the main program fragment spellcheck, together with the two main data structures \text{Word} and \text{WordList}:

Type \text{Word} has specification: \text{Word}.
Type \text{WordList} has specification: \text{seq} \text{ Word}.

Fragment spellcheck(s:WordList) has specification:
output \text{x:WordList} such that \text{elements}(x) = \text{remove}(s, \text{dictionary}).

Note that \text{spellcheck} is loosely specified: we do not mind in what order the misspelt words appear. (Leaving the operation loosely specified in this way gives the software engineer more freedom in program design, and can significantly ease the subsequent verification task.)

4.4 1st development step - implementing spellcheck

The first step in implementing spellcheck involves sorting the list and removing any correctly spelt words. We could do this in steps by first retrieving a general template for list sorting from the library (Fig. 2), which in turn includes a template which gives specifications of a number of useful list processing fragments (Fig. 3). Note that these general templates contain specifications of useful components together with their associated theory, but do not make any commitment to particular data structures or particular algorithms for sorting: there are other, more specific templates for such things.

To apply the template to our problem we would issue the following script command:

\begin{verbatim}
    instantiate General Sort with
      formal parameters: E \rightarrow \text{Word}, x < y \rightarrow \text{before}(x, y)
      textual parameters: Element \rightarrow \text{Word}, List \rightarrow \text{WordList}, increasing \rightarrow ordered
\end{verbatim}

\text{(Textual parameters refer to the names of components of the template, which can be changed to suit the problem at hand.) The effect of such an instantiation would be to bring specifications of fragments sort, emptylist, etc onto our worksheet, together with the instantiated applicability condition for the template:}

\[ \forall a, b : \text{Word} \bullet \text{before}(a, b) \lor a = b \lor \text{before}(b, a); \]
\[ \forall a, b, c : \text{Word} \bullet \text{before}(a, b) \land \text{before}(b, c) \Rightarrow \text{before}(a, c) \]

\text{(These proof obligations can be discharged by direct appeal to the definition of \text{before} above.)}

The next step would be to declare new fragments \text{dictionary} and \text{removeOKWords} corresponding to the dictionary and the operation for removing words from a list:
Template General Sort is

\textbf{include template Basic Lists}

\textbf{formal parameters: } \_ \prec \_ : E \times E

\textbf{applicability conditions:}
\begin{align*}
\forall a, b : E & \implies a < b \lor a = b \lor b < a \\
\forall a, b, c : E & \implies a < b \land b < c \Rightarrow a < c \\
\forall a : E & \implies \neg (a < a)
\end{align*}

\textbf{Theory Definition of increasing}
\begin{align*}
\text{\textit{increasing}} : & \text{seq } E; \\
\text{\textit{increasing}}(()) & \\
\forall h : E, t : & \text{seq } E \implies \text{\textit{increasing}}(\text{append}(h, t)) \iff t = () \lor (h < \text{head } t \land \text{\textit{increasing}}(t))
\end{align*}

\textbf{Theory Definition of sort}
\begin{align*}
\text{\textit{sort}} : & \text{seq } E \rightarrow \text{seq } E; \\
\forall s : & \text{seq } E \implies \text{\textit{elements}}(\text{\textit{sort}}(s)) = \text{\textit{elements}}(s), \\
\forall s : & \text{seq } E \implies \text{\textit{increasing}}(\text{\textit{sort}}(s))
\end{align*}

\textbf{Fragment sort(s:List) has specification:}
\begin{itemize}
\item output \texttt{r:List} such that \texttt{r = sort(s)}.
\end{itemize}

\textbf{Branching fragment compare(x,y:Element) has specification:}
\begin{itemize}
\item if \texttt{x < y} then report \texttt{before}
\item elseif \texttt{y < x} then report \texttt{after}
\item else report \texttt{same}.
\end{itemize}

\textbf{Figure 2: General template for sorting lists}
Template Basic Lists is

formal parameters: \( E \)

Type \( \text{Element} \) has specification: \( E \).
Type \( \text{List} \) has specification: \( \text{seq} \ E \).

Fragment \( \text{emptyList()} \) has
specification:
output \( s: \text{List} \) such that \( s = \langle \rangle \).

Fragment \( \text{append}(h: \text{Element}, t:\text{List}) \) has
specification:
output \( s: \text{List} \) such that \( s = \text{append}(h, t) \)

Fragment \( \text{concat}(s, t:\text{List}) \) has
specification:
output \( r: \text{List} \) such that \( r = s \circ t \).

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

Figure 3: A basic template for lists

Fragment \( \text{dictionary()} \) has
specification:
output \( r: \text{WordList} \) such that \( r = \text{dictionary} \).

Fragment \( \text{removeOKWords}(s, t:\text{WordList}) \) has
specification:
precondition \( \text{ordered}(s) \land \text{ordered}(t) \)
output \( r: \text{WordList} \) such that \( \text{elements}(r) = \text{remove}(s, t) \).

Note that the design assumption that the two lists to be compared are ordered has been recorded as a precondition on \( \text{removeOKWords} \): this assumption will be important for developing an efficient implementation.

Once the specifications of \( \text{dictionary} \) and \( \text{removeOKWords} \) have been brought onto the worksheet, we can implement \( \text{spellcheck} \) by issuing the following script command:

\[
\text{Fragment } \text{spellcheck}(s: \text{WordList}) \text{ has}
\text{implementation:}
\text{return } \text{removeOKWords}(\text{sort}(s), \text{dictionary}) .
\]

The proof obligations that check the correctness of this implementation are as follows:

\[
\forall s : \text{seq } \text{Word} \bullet \text{ordered}(\text{sort}(s));
\text{ordered}(\text{dictionary});
\forall s, u, v, r : \text{seq } \text{Word} \bullet u = \text{sort}(s) \land v = \text{dictionary} \land \text{elements}(r) = \text{move}(u, v)
\Rightarrow \text{elements}(r) = \text{move}(s, \text{dictionary})
\]

They are straightforward to discharge from definitions and simple properties of sequences.
4.5 2nd development step - implementing sort

Suppose we decide to implement the sorting fragment by retrieving a template for the Quick-sort algorithm from the library (Fig. 4):

- Instantiate Quicksort with
- formal parameters: $E \rightarrow \text{Word}$, $x < y \rightarrow \text{before}(x, y)$
- textual parameters: $\text{qsort} \rightarrow \text{sort}$, $\text{Element} \rightarrow \text{Word}$, $\text{List} \rightarrow \text{WordList}$.

This has the effect of bringing theory and fragments from the template onto the worksheet.

4.6 3rd development step - implementing WordList

Suppose we next decide to implement word-lists using primitive components from the library which give access to target-language data structures for linked lists (details omitted for reasons of space). Issuing the appropriate instantiation command would bring the following primitive components onto the worksheet:

Type $\text{WordList}$ has
- specification: seq $\text{Word}$
- associated code:
  
  
  "type $\text{WordList} = \left\{ \text{Cell};
  \text{Cell} = \text{record current:Word; next:WordList; end} \right\}."

Fragment $\text{emptylist}()$ has
- specification:
  output $s: \text{WordList}$ such that $s = ()$
- associated code:
  "nil".

Fragment $\text{append}(h:\text{Word}, t:\text{WordList})$ has
- specification:
  output $s: \text{WordList}$ such that $s = \text{append}(h, t)$
- associated code:
  "new s; s$.current:=h; s$.next:=t".

Together with primitive fragments for $\text{concat}$ and $\text{decomposeList}$.

4.7 4th development step - implementing removeOKWords

Turning our attention to the $\text{removeOKWords}$ fragment, we could use an “accumulator” strategy (see [23] for more explanation) to write an implementation which steps through the two lists in parallel, accumulating the words which are in the first but not in the second. This would be achieved by issuing script commands to implement $\text{removeOKWords}$ as follows:

Fragment $\text{removeOKWords}(s, t: \text{WordList})$ has
- implementation:
  return $\text{shuffleProcess}(s, t, \text{emptylist})$. 
Template Quicksort with duplicates removed is

**Theory Definition of lessThanElems and gtrThanElems**

lessThanElems, gtrThanElems : \( E \times \text{seq} E \rightarrow \text{seq} E \);

lessThanElems(e, \{\}) = \{\} = gtrThanElems(e, \{\}),

\( \forall h, c : E, t : \text{seq} E \rightarrow c < e \Rightarrow \text{lessThanElems}(e, \text{append}(h, t)) = \text{append}(h, \text{lessThanElems}(e, t)) \),

\( \forall h, c : E, t : \text{seq} E \rightarrow \neg c < e \Rightarrow \text{lessThanElems}(e, \text{append}(h, t)) = \text{lessThanElems}(e, t) \),

\( \forall h, c : E, t : \text{seq} E \rightarrow c < h \Rightarrow \text{gtrThanElems}(e, \text{append}(h, t)) = \text{append}(h, \text{gtrThanElems}(e, t)) \),

\( \forall h, c : E, t : \text{seq} E \rightarrow \neg c < h \Rightarrow \text{gtrThanElems}(e, \text{append}(h, t)) = \text{gtrThanElems}(e, t) \).

**Fragment qsort(s:List)** has

specification:

output \( r: \text{List} \) such that \( r = \text{sort}(s) \)

implementation:

cases decomposeList(s) of:

  empty: return emptylist

  nonempty: assign output to h:Element, t:List ;
              assign splitList(h, t) to lt:List, gt:List;
              return concat(qsort(lt), append(h, qsort(gt)))).

**Fragment splitList(e:Element, s:List)** has

specification:

output ll, rl:List

such that \( ll = \text{lessThanElems}(e, s) \land rl = \text{gtrThanElems}(e, s) \)

implementation:

return splitListAcc(e, s, emptylist, emptylist).

**Fragment splitListAcc(e:Element, s:List, lt:List, gt:List)** has

specification:

output ll, rl:List

such that \( ll = \text{lessThanElems}(e, s) \cap \text{lt} \land rl = \text{gtrThanElems}(e, s) \cap \text{gt} \)

implementation:

cases decomposeList(s) of:

  empty: return lt, gt

  nonempty: assign output to h:Element, t:List;
              cases compare(h, e) of:
              before: return splitListAcc(e, t, append(h, lt), gt)
              after: return splitListAcc(e, t, lt, append(h, gt))
              same: return splitListAcc(e, t, lt, gt)

variant \#s.

Figure 4: Template for the Quicksort algorithm
shuffleProcess is specified and implemented by hand by issuing the following script command:

```
Fragment shuffleProcess(s, t, w: WordList) has
  specification:
  precondition ordered(s) \land ordered(t)
  output r: WordList such that elements(r) = elements(w) \cup remove(s, t)
  implementation:
  cases decomposeList(s) of:
  empty: return w
  nonempty: assign output to a: Word, u: WordList;
  cases decomposeList(t) of:
  empty: return concat(s, w)
  nonempty: assign output to b: Word, v: WordList;
  cases compare(a, b) of:
  before: assign append(a, w) to q: WordList;
  return shuffleProcess(u, t, q)
  after: return shuffleProcess(s, v, w)
  same: return shuffleProcess(u, v, w)
```

variant \( \#s + \#t \).

The proof obligation to show that the implementation of removeOKWords is correct is:

\[
\forall s, t, w : \text{seq Word} \bullet w = \langle \rangle \land \text{elements}(r) = \text{elements}(w) \cup \text{remove}(s, t) \\
\Rightarrow \text{elements}(r) = \text{remove}(s, t)
\]

which is straightforward. The verification of shuffleProcess is outlined in the following section.

### 4.8 Verification of shuffleProcess

The partial correctness condition for the first path through the implementation of shuffleProcess is

Proof obligation shuffleProcess_pc_path1

```
== \forall s, t, w, r : \text{seq Word} \bullet \text{ordered}(s) \land \text{ordered}(t) \land s = \langle \rangle \land r = w \\
\Rightarrow \text{elements}(r) = \text{elements}(w) \cup \text{remove}(s, t)
```

This checks that, when \( s \) is empty (\( s = \langle \rangle \) ) and the output is \( w \) (\( r = w \) ) then the desired input/output relationship is achieved. This proof obligation can be discharged by arguing that \( \text{remove}(s, t) = \emptyset \) and \( \text{elements}(w) \cup \text{remove}(s, t) = \text{elements}(w) \), as required.

The other partial correctness proof obligations are similar and not much more difficult to prove.

As an example of a well-formedness condition, here is the proof obligation which checks the precondition for the recursive call to shuffleProcess in the third path:

Proof obligation shuffleProcess_wff_path3

```
== \forall s, t, u, v, q : \text{seq Word}, a, b : Word \bullet \\
  \text{ordered}(s) \land \text{ordered}(t) \land s \neq \langle \rangle \land s = \text{append}(a, u) \land t \neq \langle \rangle \\
  \land t = \text{append}(b, v) \land \text{before}(a, b) \land q = \text{append}(a, w) \\
\Rightarrow \text{ordered}(u) \land \text{ordered}(t)
```

The result follows by simple reasoning from the fact that the tail of an ordered list is itself an ordered list. (Such a fact could be proved using one of the theorem provers and added to the theory of ordered sequences as a lemma.)
type Word = { code for Word };  
WordList = |Cell; 
Cell = record  
current: Word;  
next: WordList;  
end;  

function spellcheck(s: WordList): WordList;  
var r: WordList;  
begin  
r := shuffleProcess(sort(s), dictionary(), nil);  
spellcheck := r;  
end;  

...  

function shuffleProcess(s, t, w: WordList): WordList;  
var a, b: Word, r, q: WordList;  
begin  
if s=nil then r := w  
else  
begin  
a := s[].current;  
b := t[].current;  
v := t[].next;  
if {code to test before(a,b)} then  
begin  
new q; q[].current := a;  
q[].next := w;  
r := shuffleProcess(u, t, q)  
end  
else if {code to test before(b, a)} then {...}  
else {...}  
end  
shuffleProcess := r;  
end

Figure 5: Synthesized code for the spellchecker system

4.9 Synthesized code

Fig. 5 gives a (partially elided) listing of the Pascal code that would be synthesized from the development as given so far. To complete the development it would be necessary give implementations of the CARE type Word and the fragments compare and dictionary.

5 Comparison with other approaches

The CARE approach to formal software development is based on the model-oriented, hierarchical development approach epitomized by VDM [17]. Abstract high-level specifications are progressively transformed into low-level executable specifications through a stepwise process adding algorithmic detail and refining data representations.

To the best of our knowledge, no comparable toolset is available for VDM or Z. The murnal environment [16] supports data refinement in VDM and reasoning about specifications, but it does not support algorithm refinement or translation to code. The IFAD VDM-SL Toolbox [19] can be used for prototyping and executing VDM specifications but it does not support formal verification. KIDS/VDM [20] supports prototyping of VDM specifications through soundness-preserving transformations but falls short of general support for refinement. There are various tools for reasoning about Z specifications (e.g. ProofPower [3]) but none support general refinement, with the notable exception of Cogito [9], which uses
a VDM-like approach to refinement; however, Cogito makes little attempt to hide its full mathematical machinery from the user and has not yet explored reuse of designs.

B is a model-oriented development method broadly similar to VDM. Experience with B [7] seems to indicate that, like Care, it uses a simpler approach to refinement than VDM, but that it would still be considerably more difficult to learn to apply than Care.

KIDS [28] is a semi-automated system for transforming executable specifications into efficient programs in a soundness-preserving manner, with user selection from high-level options. KIDS is being incorporated into a development support system at the Kestrel Institute [18] which is broadly similar to Care in its aims: the main differences seem to be that Care gives the user more control over development and verification through the use of interactive (as opposed to semi-automatic) tools, and Care has a broader framework; however, the (as yet unnamed) Kestrel system has more advanced CASE features.

Another related system is AMPHION [30] which makes use of a library of formally-specified FORTRAN routines. AMPHION converts space scientists’ graphical specifications into mathematical theorems and uses automated deduction to try to construct and verify a program that satisfies the specification. The success of AMPHION in its particular problem domain gives us further reason to believe that our approach to using library routines is feasible.

We believe that Care offers better chance of scaling up than the Refinement Calculus [27] or the more traditional VCG-like approach to program development of systems such as EVES [12], since it supports hierarchical development and reusable design templates. We have shown that Care is “expressively equivalent” to the Refinement Calculus in that rules from the latter can easily be expressed as Care templates and vice-versa [21].

Finally, we note that Care programs have a strong “functional programming” flavour. Note however that Care delivers compilable code for procedural programming languages, which makes it more widely usable for application software than a functional programming language.

6 Conclusions

In summary, the Care toolset helps software engineers develop formally verified programs from abstract formal specifications. The tools have been developed in response to identified industrial needs for a formal software development method which does not require the user to be an expert in formal proof. Our experience training novice users has shown that Care can be used on small examples after only a basic course in formal specification. The tools help the user build applications by selecting and instantiating pre-proven refinements to fit the problem at hand, and generating and discharging correctness-of-fit proof obligations. The correctness of each design step can be checked immediately, thereby significantly shortening the feedback loop. Provision of reusable design templates means that most of the difficult parts of a proof can be done once, off-line by suitably skilled experts, leaving only proof of the applicability conditions to the software engineer.

Another major innovation of the Care approach is that it gives a means for developing verified software for target languages which themselves do not have a formal semantics. It achieves this by restricting the user’s use of target-language code to formally specified library routines which are verified “off-line”. The Care code synthesis process relies on the target language for little more than simple sequencing, assignment to local variables, alternation (if-then-else) and recursion — all of which are well understood for most commonly used programming languages.
Acknowledgements: The authors would like to thank their colleagues on the CARE project, past and present, especially Keith Harwood at Teletronics who has been developing many of the ideas underlying CARE over many years. We would also like to thank our other colleagues at Teletronics and the SVRC for many fruitful discussions.

References


