SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-13

An industrial-strength method for the construction of formally verified software

Peter A. Lindsay and David Hemer

June 1996

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`.

An industrial-strength method for the construction of formally verified software *

Peter Lindsay and David Hemer
email: pal,hemer@cs.uq.edu.au

To appear: *Proceedings 9th Australian Software Engineering Conference (ASWEC'96)*, Melbourne, July 1996, IEEE Computer Society Press.

Abstract

The CARE method is a new approach to constructing and formally verifying programs. CARE has 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. Software engineers use CARE to develop compilable code from formal program specifications using a library of pre-proven, formally specified refinements. Tools help users build products by selecting and instantiating refinements to fit the problem at hand, and generating and discharging correctness-of-fit proof obligations.

This paper introduces CARE's integrated notation for algorithm specification and development, and explains how correctness is checked. The method is illustrated on a small development.
Keywords: formal methods, program development, software verification, refinement

1 Introduction

1.1 Motivation

There is a growing demand for industrial-strength formal methods for software development. Many companies now see the benefits of formal specification techniques and are using them to specify parts of their systems or to document their code [11]. Formal specification offers improved understanding of systems under development and formal validation techniques allow cross-checks to be performed on specifications, thereby discovering and fixing mistakes early in the development life-cycle [5, 6].

Government regulatory and standards authorities are increasingly mandating the use of formal methods in the development of critical software: examples are the Department of Defense and the National Computer Security Center in the USA, the Ministry

*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.

of Defence [3] and the Communications Electronics Security Group in the UK, the International Electrotechnical Commission of the European Union [4], and the Defence Signals Directorate and the Ordnance Council in Australia. In fact, the Australian Ordnance Council's Pillar Proceeding 223.93 [2] even mandates the use of formal methods for safety-critical software.

Formal refinement techniques extend the benefits of formal specification further into the development process by enabling designs and implementations to be expressed formally. In theory at least, formal verification can then be used to check that implementations satisfy their specifications. As currently practised, however, formal verification is labour intensive and requires highly specialised skills. Experience has shown that industrial software engineers are often proficient in the area of product development but not as good at formal specification and proof.

1.2 The CARE method

The CARE method [14, 16, 24] was designed to enable industrial software engineers to develop formally verified software from formal specifications. The method was developed through 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 provably correct software. A grant from the Australian Government has enabled more extensive development of Keith Harwood's original ideas [13] and the construction of a prototype toolset to support the method.

CARE stands for **Computer Assisted Refinement Engineering**. In the software development life-cycle, CARE is used after system requirements have been defined and analysed for criticality, and system requirements have been mathematically modelled by writing a formal specification. Starting with the formal specification, software engineers use CARE to develop compilable code, with integrated checking that the code is provably correct. The CARE method can be adapted to deliver source code in most common programming languages.

The process of developing software using CARE involves selection of refinements from a library of pre-proven parameterized *templates*, and instantiating them to suit the problem at hand. The end-result is a CARE program, from which a target-language-specific source-code program can be mechanically synthesized.

1.3 Verification in CARE

The CARE verification and synthesis process is illustrated in Fig. 1 and explained in more detail in Sections 2–3; here we simply give a brief overview.

Each component of a CARE program has a formal *specification* which characterizes the component mathematically, together with an *implementation*. *Primitive* components are implemented directly in the target language and are supplied with the CARE library; CARE users do not write target code directly. *Higher-level* components are implemented by combining calls to other components in a special-purpose language with

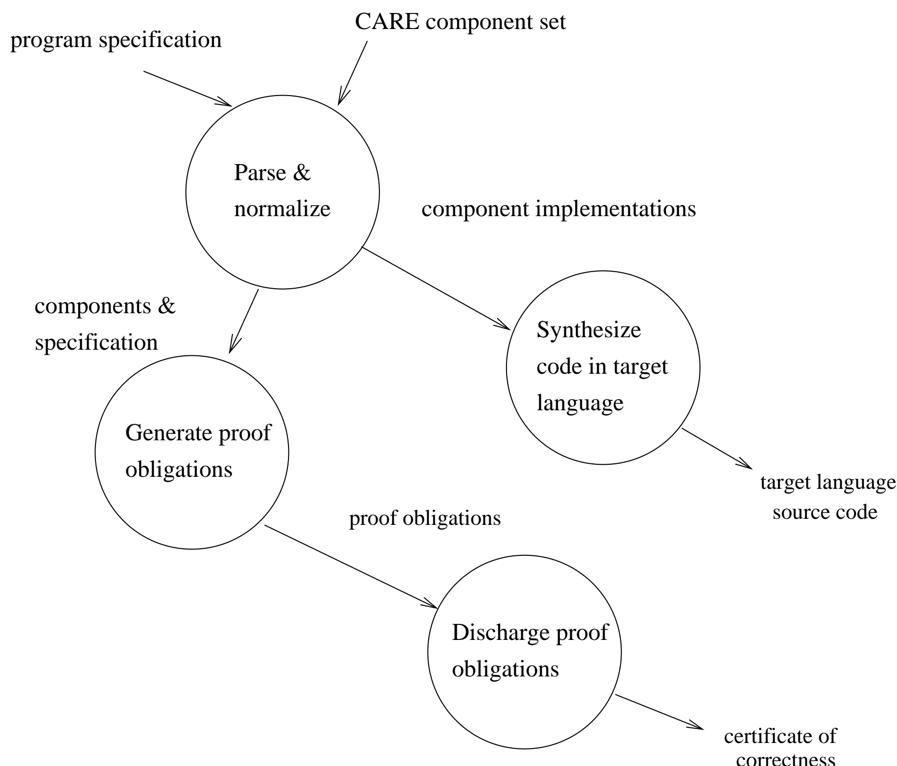

Figure 1: The checking and synthesis process for CARE-generated programs.

a simple, formally defined mathematical semantics; the implementations are either written directly by the user or are selected from library templates.

Proof obligations can be generated mechanically from the CARE program, to check that templates' applicability conditions are satisfied and that implementations of higher-level components satisfy their specifications. The engineer's proof obligations can be kept to a minimum by making use of templates which have been proven off-line by verification experts and which are supplied with the CARE library.

When a CARE program is complete (i.e., all components have been implemented), a target-language-specific CARE tool can be used to mechanically synthesize a complete source-code program. If all of the proof obligations have been discharged, the synthesized program is guaranteed to satisfy its specification — assuming of course that the primitives have been modelled appropriately and that the compiler is correct.

A prototype tool-set has been developed to support the CARE method (see Fig. 2). The tool-set includes parsers, syntax- and type-checkers, pretty-printers, documentation preparation tools, proof obligation generators, mathematical simplifiers, an automatic theorem prover, automated support for formal reasoning in an interactive theorem prover, and a code synthesizer with C as the target language. A large library of pre-proven templates has been produced, together with a tool which assists software engineers in selecting and instantiating templates.

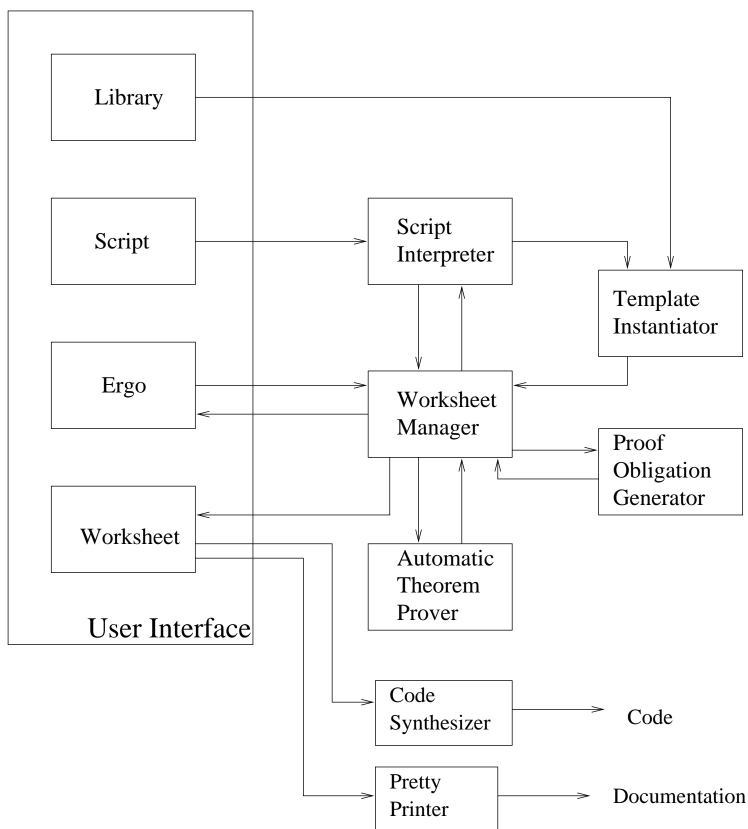

Figure 2: The CARE prototype tool-set.

1.4 Using CARE in system development

The CARE method is largely independent of the particular specification notation, target language, compiler and platform used. The current prototype uses Z [28] as its mathematical specification notation 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 [12]. 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. Because they are target-language source-code programs, CARE-synthesized programs can be integrated with other system components and tested using traditional integration techniques. 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.

CARE can also be used in conjunction with formal development techniques such as VDM [18] or Morgan's Refinement Calculus [26]. In part, CARE can be seen as a way of giving further structure to VDM or Refinement Calculus developments — structure which is useful for raising the level at which one reasons about designs and design choices, while hiding lower-level verification aspects.

In 1995 the CARE method was trialled at Telectronics in a five-day intensive training course attended by senior software engineers not directly involved in the CARE project. Telectronics have now decided to use the approach to develop part of the software for its next product range.

1.5 Comparison with related projects

The CARE approach to formal software development is based on the model-oriented, hierarchical development approach epitomized by VDM [18] and B [20], in which 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 tool-set is available for VDM or Z. The `mural` environment [17] 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 [21] can be used for prototyping and executing VDM specifications but it does not support formal verification. KIDS/VDM [22] 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 [1]) but none support general refinement, with the notable exception of `Cogito` [8], 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, but with better support for modularity in specification and implementation. 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, and its support for verification seems to be not as effective. The SVRC has recently purchased the B-Toolkit and we shall evaluate its effectiveness during 1996.

KIDS [27] 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 [19] which is broadly similar to CARE in its aims: the main difference is 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 Kestrel system has more advanced CASE features.

Another related system is AMPHION [29], 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 is further evidence that the CARE approach to using library routines is effective. The CARE problem domain is however far wider than that of AMPHION, and domain theories are far less developed.

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 [23]. However, CARE has constructs (such as design templates and recursive fragments) which make it better able to support reuse and component-wise verification, and for which better user support can be provided.

Finally, we note that CARE can deliver compilable code for most programming languages, which makes it more widely usable in software engineering than many other methods.

1.6 This paper

The rest of this paper introduces CARE's integrated notation for algorithm specification and development (Section 2), explains the proof obligations which check correctness (Section 3), and illustrates the method on a small application (Section 4).

2 The CARE notation and semantics

2.1 Overview

A CARE program consists of mathematical definitions and lemmas, together with a set of CARE *components* known as *types* and *fragments*. Types are used for defining data structures and fragments are used for defining algorithms.

Each CARE component has a formal specification and an implementation. The specification of components is expressed using an extended form of the Z mathematical toolkit presented in [28, 9].

Components of a CARE program can be categorised as either *primitive* or *higher-level* components. *Primitive components* provide access to target language data structures and basic functionality, and are provided to the CARE user as a library. Primitive components are implemented directly in the target language. The specification of a primitive component describes the component in terms of (a mathematical model of) the semantics of the target language and its compiler. A primitive type's specification describes the set of mathematical values corresponding to the associated data structure. A primitive fragment's specification describes the associated target code's functionality. *Higher-level components* express data refinements and algorithm designs, and are written in a special purpose language. The CARE language supports the following simple design constructs: assignment of values to local variables, fragment calls, sequencing, branching of control, recursion, and data refinement transformations.

Sections 2.2-2.4 below describe aspects of the CARE language in more detail. (The notation described here is a verbose form used for didactic purposes; the prototype tools use a terse form better suited to mechanical manipulation.) In the rest of this paper, CARE values and types are written in `typewriter` font and mathematical expressions are written in *italics* using the Z notation.

2.2 Types

CARE types consists of an identifier, a specification and an implementation. The

specification of a CARE type defines the set of values that objects of this type may take. The implementation of a CARE type may be either *primitive* (i.e. implemented directly as a target language data structure) or *higher-level* (used to express data refinements). Space limitations do not permit a full treatment of type implementations here.

Below are examples of unimplemented (specified only) types:

> Type `Element` has specification: $ELEMENTS$
> Type `List` has specification: $\mathrm{seq}\, ELEMENTS$
> Type `Natnum` has specification: $\mathbb{N}$

`Element` corresponds mathematically to a set of elements (not specified any further here), the type `List` corresponds to the collection of sequences of elements, while `Natnum` corresponds to the set of natural numbers.

2.3 Fragment specifications

In describing the specifications of fragments we shall consider the two classes of fragments - *simple* and *branching* - separately.

The specification of a *simple fragment* defines the fragment's inputs and their types, the precondition, the number and types of outputs, and the required input/output (I/O) relationship (or postcondition). The precondition expresses constraints on which inputs can be supplied to a fragment. If a fragment is called outside its precondition, no guarantee can be placed on its actions. (Proof obligations will check that fragments are only ever called on arguments which satisfy the fragments' preconditions).

The fragments `cons`, `car` and `cdr` used for list manipulation are given below. The fragment `cons` takes an element, `e`, and a list, `s`, and forms a new list by appending `e` onto the left end of `s`. The fragments `car` and `cdr`, which both take a list, `s`, as input return the *head* of `s` (the left-most element of `s`) and the *tail* of `s` (the remainder of the list `s` after the left-most element is removed) respectively. Both fragments have a precondition, which constrains the input to those lists which contain at least one element:

> Fragment `cons(e:Element,s:List)` has
> specification:
>   output `r:List` such that $r = \langle e \rangle \frown s$.
>
> Fragment `car(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `h:Element` such that $h = head\, s$.
>
> Fragment `cdr(s:List)` has
> specification:
>   precondition $\#s \neq 0$
>   output `t:List` such that $t = tail\, s$.

The specification of a branching fragment consists of a description of the inputs and their types, an optional precondition and a sequence of guarded branches. Each branch

contains a description of the outputs and their types, an I/O relationship and a report. The report uniquely identifies each branch of the specification and can be used in fragment implementations to refer to a particular branch. For each guarded branch the guard defines the set of input values for which the following branch can be used to define the result.

For example, the fragment `decomposeList` given below has two different cases: one when the list is empty (in which case it reports `empty` with no outputs), and the other when it is non-empty (in which case it returns the head and the tail of the list and reports `non-empty`):

> Branching fragment `decomposeList(s:List)` has
> specification:
>   result defined by cases
>   if $\#s = 0$ then report `empty`
>   else report `nonempty`
>   with output `h:Element, t:List` such that $s = \langle h \rangle \frown t$.

2.4 Fragment implementations

A primitive fragment is implemented in terms of target language code (not treated further here). A higher-level fragment is implemented in terms of calls to other fragments. The body of a higher-level fragment is tree-structured. Non-branching nodes of the tree correspond to bindings to local variables of the values returned by simple fragment calls and/or variables. Branching nodes correspond to calls to branching fragments; where branches return values, these values are bound to local variables. The leaves of the tree define the fragment's output values. For branching fragments, the leaves also contain a report, referring to one of the branches of the specification. A variant may also be given for higher-level fragments, which is a natural-number-valued function of the input values, used to establish termination of recursive calls. An *abort* statement is provided for use in branches which will never be executed.

For example, consider the branching fragment `null`, which tests whether or not a list is empty:

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

The fragment `decomposeList`, specified earlier, can be implemented by testing whether the list is empty via a call to the branching fragment `null`, and if it is not, then calling the fragments `car` and `cdr` to return the head and tail of the list:

```
Branching fragment decomposeList(s:List) has
implementation:
   cases null(s) of:
      yes: report empty
      no:  report non-empty and return car(s),cdr(s).
```

3 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.

Using the CARE method, verification of a fragment set involves establishing a number of *proof obligations*, which fall into four categories:

Well-formedness: For each fragment call in an implementation tree, the called fragment's precondition (if any) must be satisfied at that point.

Partial correctness: The result returned at each (non-aborting) leaf of an implementation tree must satisfy the appropriate I/O relationship.

Termination: For recursively defined fragments, the variant must be a $\mathbb{N}$-valued function and must be strictly decreasing on recursive calls. (Since the variant is bounded below by zero, it cannot decrease indefinitely, so the recursion will therefore terminate in a finite number of steps.)

Non-execution: Execution must not be able to 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 following sense: execution of one of the fragments on input values which satisfy its precondition will terminate and return a result which satisfies the fragment's specified I/O relationship.

In practice, 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). The proof obligation involves showing that the desired conclusion follows from assumptions about the execution path to the point in question. For each fragment call along the path, the appropriate I/O relationship is assumed to hold. For paths which pass through branch points, the appropriate guards are also assumed to hold. Finally, the calling fragment's precondition is assumed to hold.

For example, we shall consider the partial correctness proof obligation for the leaf of the second path through `decomposeList` given by:

```
case no of null(s);
report non-empty and return car(s),cdr(s)
```

Since the report (`non-empty`) of this path means that the second branch of the specification describes the desired result, we want to show that the guard ($\#s \neq 0$) and the I/O relationship ($s = \langle h \rangle \frown t$) for the second branch of the specification holds. To prove this we can assume the precondition and I/O relationships of the fragments `car` and `cdr`, as well as the guard of the second branch of the fragment `null`. This leads to the following proof obligation:

$$
\begin{array}{l}
\forall\, s : \mathrm{seq}\, \mathit{ELEMENTS} \bullet \\
\quad \neg\, (\#s = 0) \Rightarrow \\
\qquad \forall\, h : \mathit{ELEMENTS} \bullet h = \mathit{head}(s) \Rightarrow \\
\qquad\quad \forall\, t : \mathrm{seq}\, \mathit{ELEMENTS} \bullet t = \mathit{tail}(s) \Rightarrow \\
\qquad\qquad \neg\, (\#s = 0) \wedge s = \langle h \rangle \frown t
\end{array}
$$

For a more detailed description of proof obligations and how they are generated, the reader is referred to [15].

4 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 [26]. The algorithm is developed through a series of design choices and each stage in the design is verified before passing to the next stage.

4.1 Program specification

The program has fragment specification

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

4.2 First design step

The first step in a development of a program to satisfy this specification might be to introduce 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 `increment(s)` to `hi:Natnum`;
>   return `iterate(s,lo,hi)`.

where

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

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

are fragments one could expect to find as primitives 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(increment(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 user-defined fragment which performs the necessary iteration using an auxiliary fragment `closeGap` (specified below), together with

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

which again one could expect to find as a primitive in the library. `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`.

4.3 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 `increment`, well-formedness of the call to `iterate` from `sqroot` involves showing

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

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 implementation 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

$$(\underbrace{lo < hi \wedge lo^2 \leq s < hi^2}_{precondition} \wedge \underbrace{lo+1 \not< hi}_{\texttt{no}\ case}) \Rightarrow \underbrace{lo^2 \leq s < (lo+1)^2}_{output\ condition}$$

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.

4.4 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: $lo < mid < hi \wedge lo^2 \leq s < hi^2$
>   output `u,v:Natnum` such that $u^2 \leq s < v^2 \wedge 0 \leq v - u < hi - lo$.
> implementation:
>   cases `lessthan(s,square(mid))` of
>   `yes`: return `lo,mid`.
>   `no`: return `mid,hi`.

are user-defined fragments and

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

would be another library fragment.

The two partial correctness proof obligations for `adjustBnds` are

$$(lo < mid < hi \wedge lo^2 \leq s < hi^2 \wedge s < mid^2) \\ \quad \Rightarrow (lo^2 \leq s < mid^2 \wedge 0 \leq mid - lo < hi - lo)$$

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

The other proof obligations are easy to check.

4.5 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 $n = m \text{ div } 2$.

4.6 Summary

This completes the development of `sqroot`. Fig. 3 shows the Pascal program that would be synthesized from this design; the actual code synthesized by CARE tools would obviously depend on what target language was used.

The example has illustrated the CARE notation for algorithm design and the use of proof obligations to verify algorithms. For an example of program design involving more complicated data structures (including the use of data refinement) see [25]. For an example of the use of design templates in program construction, see [16].

```
Program sqroot(s:Nat)
  var lo,hi:Nat;
  label 1;

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

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

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

5 Conclusions

This paper has outlined the CARE approach to constructing and verifying software, concentrating on the development of algorithms from formal program specifications. Using the CARE approach, the software engineer assembles a collection of CARE components and applies mechanized analysis tools to check the correctness of the combination against specified requirements. The tools check that the components fit together properly and achieve the desired overall effect. Automated theorem provers are used to discharge the proof obligations that arise. 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 notation allows the software development process to be structured in such a way that engineering aspects (such as requirements specification, algorithm design, and choice of data structures) are separated from formal mathematical aspects (such as proof obligation generation and proof of correctness). As far as possible, formal verification aspects of the method are consigned to automated tools, allowing the software engineer to concentrate on the design and development of useable, efficient pieces of software. CARE has been applied, for example, to check the logic of an event logger similar to the kind found in certain medical embedded devices [25].

The method is general and can be used in conjunction with a variety of other development methods, both formal and informal. It can be used with a wide variety of specification languages, theorem provers and target languages. Finally, CARE has been shown to be industrially useable by trialling it with software engineers in Telectronics who were not directly involved in its development.

Acknowledgements:

The authors would like to thank their colleagues on the CARE project, including Keith Harwood, Thies Arens, Frances Collis, Rex Matthews and Trudy Weibel. Thanks also for the constructive comments made by many of our colleagues at Telectronics and the SVRC, including in particular John Staples and Ian Hayes.

SVRC technical reports are available by anonymous ftp from ftp.cs.uq.edu.au in the directory /pub/SVRC/techreports.

References

[1] ProofPower home page. http://www.to.icl.fi/ICLE/ProofPower/index.html.

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

[3] The Procurement of Safety Critical Software in Defence Equipment. U.K. Ministry of Defence, (draft) May 1995. Defence Standard 00-55.

[4] Functional safety: safety-related systems. International Electrotechnical Commission, June 1995 (draft). International Standard IEC 1508.

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

[6] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.

[7] J. C. Bicarregui and B. Ritchie. Invariants, frames and postconditions: a comparison of the VDM and B notations. In *FME'93: Industrial Strength Formal Methods*. Springer Verlag, 1993. Proc. First Internat. Symp. of Formal Methods Europe, Odense, Denmark, April 1993.

[8] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: A Methodology and System for Formal Software Development. *International Journal of Software Engineering and Knowledge Engineering*, 5(4), December 1995.

[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] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, pages 21–28, January 1994.

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

[13] K. 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.

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

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

[16] D. Hemer and P.A. Lindsay. The CARE toolset for developing verified programs from formal specifications. In *Proc. 4th IEEE Int. Symp. on Assessment of Software Tools*, 1996. Available by ftp as SVRC TR 95-52.

[17] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

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

[19] R. K. Jullig. Applying formal software synthesis. *IEEE Software*, pages 11–22, May 1993.

[20] K. Lano. *The B Language and Method.* FACIT Series. Springer-Verlag, 1996.

[21] P. B. Lassen. IFAD VDM-SL toolbox report. In *FME'93: Industrial Strength Formal Methods*, page 681. Springer Verlag, 1993. Proc. First Internat. Symp. of Formal Methods Europe, Odense, Denmark, April 1993.

[22] Y. Ledru. Proof-based development of specifications with KIDS/VDM. In *FME'94: Industrial Benefits of Formal Methods*, pages 214–232, 1994. 2nd International Symposium of Formal Methods Europe, Barcelona, October 1994.

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

[24] P.A. Lindsay. The CARE method of verified software development. Technical Report 95-9, Software Verification Research Centre, The University of Queensland, 1995.

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

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

[27] D. Smith. KIDS: a semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[28] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, New York, 1989.

[29] 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.