SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 96-23

A template-based approach to construction of verified software

Peter A. Lindsay David Hemer

October 1996

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

A template-based approach to construction of verified software

Peter A. Lindsay David Hemer

Abstract

This paper outlines a new approach to construction and verification of software, developed in response to identified industrial needs for a formal development method which does not require the user to be an expert in mathematical logic. The approach is based on a framework which allows formal verification to be performed off-line or consigned to automated tools, thereby allowing the software engineer to concentrate instead on the design and development of useable, efficient pieces of software.

1 Introduction

Formal verification of software is often regarded as a difficult, time-consuming task requiring esoteric mathematical skills. This paper introduces a new approach to the construction of formally verified software — called CARE, for **Computer Assisted Refinement Engineering** — which aims to bring formal verification within the reach of software engineers trained in formal specification, without requiring them to be experts in formal mathematics.

1.1 Motivation

Before explaining the aims of CARE in more detail, we briefly review the state of the art in formal verification of software. The starting point for verification is a formal specification of the software to be developed, and a wide variety of formal specification techniques are already in use in industry [1, 8, 24]. A number of program verification techniques have been proposed, falling into two broad camps:

- methods for transforming abstract specifications into concrete, implementation-level specifications: e.g. VDM [16], the Refinement Calculus [23], Larch [10], B [18], Deva [30] and Cogito [5]; and
- methods for reasoning directly about programs in (subsets of) certain programming languages: e.g. Spark Examiner [6], Gypsy [9] and EVES [7].

However full program verification is rare in industry.

Program verification requires good tool support since proofs are just as error-prone as programs – and indeed may be even more so, since they are typically orders of magnitude larger than the programs they purport to prove. For high degrees of assurance it is thus desirable to use tools to check the correctness of proofs, for which the proofs must be given in full formal detail. This brings us to the first main problem with industrialization of formal verification: skills and experience in constructing formal proofs are currently rare, especially in industry.

A second, perhaps more fundamental problem is the danger of faulty modelling and axiomatization. In any sizeable formal development, the software engineer will introduce new definitions to represent the artifacts to be constructed – foremost, but not exclusively, in the formal specification of the program. With unfettered use of definitions there is a strong possibility that logical inconsistencies will be introduced, even when in the hands of experienced mathematical logicians. Most of the industrially popular formal methods provide little or no support for checking the consistency of user-introduced definitions, yet since anything can be proved from inconsistent definitions, it is critically important to preclude them.[1]

The combination of the inherent dangers in formalization and the lack of appropriate proof skills in industry leads us to believe that an industrially-useable formal development method must try to reduce as far as possible the end-user's need for expertise in formal mathematics.

1.2 The CARE approach

The CARE solution is to provide a framework within which specification, programming and verification knowledge can be recorded and reused with minimal need for re-proof. The CARE project has been exploring the use of a library of "design templates" for which most of the difficult parts of modelling and proof have been done once, off-line, by suitably skilled experts. CARE tools then 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. Other CARE tools synthesize compilable source code programs which can be integrated with other system components and tested using traditional integration testing techniques.

[1]The problem of determining consistency is of course generally undecidable. Model-oriented specification techniques allow definitions to be grounded in set theory, which in theory provides a basis from which to justify mathematical consistency (soundness): cf. VDM's well-formedness proof obligations, for example [2, 3]; however, such proofs are difficult or tedious and rarely carried out in practice. (Note for example the prevalence of 'axiomatic definitions' in Z specifications, and how rarely the reasons for their soundness is documented.) Most algebraic specification methods do not directly support proofs of consistency, although the Larch Prover has automated checks for certain forms of inconsistency.

CARE is a generic method, in-as-much-as it can be tailored for use with different specification languages and different programming languages. In particular, it can be used to construct verified software for programming languages which themselves do not have a full formal semantics, by restricting use of target-language code to formally specified library routines which have been verified off-line using techniques appropriate to the target language.

CARE 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. Specifically, Telectronics had used formal specifications in the development of some of its products, but wanted a method and tools to help verify code and to enable tracing of requirements from specifications through to code and construction of product variants [11]. 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 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 [22]. Work is proceeding on populating the tools with further general purpose design templates and primitive components, developing larger case studies, and improving the functionality of the library browsing tool. The tools themselves have been formally specified [14].

1.3 This paper

This paper explains the CARE approach and illustrates some of the main concepts. Section 2 below introduces the CARE integrated specification and implementation language. Section 3 outlines how verified programs are developed using CARE. Section 4 explains the CARE language in more detail and illustrates its use for algorithm refinement. (Data refinement is omitted due to lack of space). Section 5 explains the CARE proof obligations for algorithm verification. Section 6 illustrates the use and verification of templates. Section 7 compares the approach with other methods for formal software development.

2 Overview of the CARE language

At the heart of CARE is an integrated language for software specification and development and a library of pre-verified refinement templates. A CARE development has a mathematical part and a program part. The mathematical part consists of mathematical definitions, theorems and proofs. The program part consists of *types* and *fragments*: roughly speaking, CARE types correspond to data structures, and fragments correspond to function and procedures in a procedural programming language.

Each CARE program component has its own formal specification, which may include a *precondition* which defines the circumstances under which the component may be used. Program components are "primitive" or "higher-level":

Primitive components are implemented directly in the target programming language and provide access to target language data structures and basic functionality. (B [18] uses a similar approach.) Primitives are supplied as part of the CARE library, and are not written by the ordinary user. 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. Proof of correctness of primitives' specifications is outside the scope of CARE; such proofs could for example be handled instead by one of the techniques mentioned earlier.

Higher-level components express data refinements and algorithm designs, and are implemented in a special-purpose language with 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 preconditions are satisfied, implementations achieve their specifications, and recursion terminates (see Section 5 for details).

CARE programs are developed incrementally – top-down, bottom-up or in a mixture of styles. During development there may be components which have specifications but which do not yet have implementations.

CARE *templates* are self-contained, parameterised collections of theory and program components, some of which may be specified only. Templates record programming knowledge in a largely self-contained manner, and are the main mechanism for reuse. There are templates for sets of related primitives, as well as for commonly used algorithm and data refinements. 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.

CARE tools check the syntactic and type correctness of CARE programs.

3 Verified program development using CARE

A typical CARE development might proceed as follows:

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 object to be considered, together with functions on those objects and relationships between them. The prototype CARE toolset accepts definitions written in a mathematical notation based on the Z mathematical tool-kit [25, 27] which in turn is based on many-sorted set theory. The template library includes definitions of commonly used mathematical constructs such as sets, sequences, relations and mappings. The user is encouraged to use the library as far as possible rather than developing their own (possibly inconsistent) definitions.

Program specification: The next step is to formally specify the application program, by giving CARE specifications of its types and fragments. 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 [13].

Program development: Using the CARE language, the software engineer typically develops a program design by progressively adding algorithmic detail and refining abstract data structures into more concrete representations. Bottom-up development is also possible, by implementing new components in terms of existing components. Program development eventually terminates when all components have been implemented or are primitive.

Design verification: At any stage in development, 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 prototype CARE toolset includes a fully automatic resolution-based theorem prover which discharges many of the simpler proof obligations, and the `Ergo` general-purpose interactive theorem prover [29] for the more difficult ones. The CARE library contains various general and domain-specific proof tactics for use with the interactive prover, so that CARE users can experiment with different combinations of tactics until they find proofs (or refutations) of 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 has been verified against its original specification. (Note of course that the program's overall correctness depends on a number of factors which are outside the scope of CARE, such as the original specification's consistency, and the correctness of the primitive components and the CARE synthesizer with respect to the host architecture.)

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.

4 The CARE language

This section describes the CARE language in detail. 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. (The notation described here is a verbose form used for didactic purposes; the prototype tools use a terse form better suited to mechanical manipulation).

4.1 Mathematical definitions

The mathematical theory part of a CARE program consists of signatures and definitions of constants, functions and relations, together with generic (not-further-defined) sets of values. Definitions are given as axioms and theorems. For example, Fig. 1 shows the pretty-printed version of the definition of a mathematical function *append* which appends a value onto the head of a sequence. There is a proof obligation to show that any new definitions are mathematically consistent. Mathematical theory is added to a CARE program in one of two ways: by the user, either from the original problem specification or from a concept introduced during design; or from the library, as part of an instantiated design template. The latter is the preferred option where possible.

Theory definition of *append*.

$append : Elem \times \mathrm{seq}\, Elem \rightarrow \mathrm{seq}\, Elem;$

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

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

Figure 1: Definition of a function for appending an element onto the front of a list.

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

Type `Element` has specification: $Elem$

Type `List` has specification: $\mathrm{seq}\, Elem$

Figure 2: Some example type specifications.

4.2 Types

A Care type declaration consists of an identifier, a specification and an implementation. The specification is a Z expression denoting the set of mathematical values that objects of the type can take. For example, Fig. 2 contains specifications of Care types for natural numbers, elements, and sequences of elements respectively. Type implementations will not be treated here, for space reasons.

4.3 Fragment specifications

There are two kinds of fragments: *simple* and *branching*. Simple fragments correspond roughly to functions in a procedural programming language; they take inputs and return outputs. Branching fragments differ from simple fragments by also allowing branching of control during execution. A non-standard feature of the Care language is that branching fragments can return different numbers and kinds of outputs on different branches.

The number and type of inputs taken by a fragment is fixed. The specification of a simple fragment consists of an optional *precondition*, a list of outputs and their types, and the required input/output relationship (or *postcondition*). Fig. 3 gives examples of specifications of a number of simple fragments for manipulating lists, using LISP-like naming conventions. Preconditions express constraints on the inputs which can be supplied to a fragment: e.g. the `car` fragment, for finding the head of a list, requires that the list be non-empty. Proof obligations will check that fragments are only ever called on arguments

```
Fragment nil() has specification:
    output s:List such that $s = \langle\,\rangle$

Fragment cons(e:Element,s:List) has specification:
    output r:List such that $r = append(e, s)$

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 $t = tail(s)$
```

Figure 3: Some example specifications of simple fragments on lists.

which satisfy their preconditions.

The specification of a branching fragment consists of an optional precondition and a sequence of guarded branches. Each branch contains a *test*, a description of the outputs and their types, an optional postcondition, and a *report*, which identifies the branch. (The test in the last branch is `true` by default.) Fig. 4 gives examples of specifications of branching fragments on lists. Note that the number and type of outputs on each branch is fixed but may differ from branch to branch. For example, the `search(s,x)` fragment has two cases: when `x` occurs in `s`, it reports `found` and returns an index `i` at which `x` can be found; otherwise it simply reports `notfound` with no outputs. The *guard* of a branch is the test conjoined with the negations of the tests of the preceding branches. For example the guard of the nonempty branch of `decomposeList(s)` is $\neg\,(\#s = 0)$.

Note that fragment specifications may be under-determined, in the sense that more than one output may satisfy the postcondition for any given input (e.g. `i` in `search(s,x)`). In practice, the postcondition is often an equation defining the output variables directly as a function of the input variables. The Care tools put this observation to good use, as shown below.

4.4 Fragment implementations

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

In effect, the body of a higher-level fragment is tree-structured. Non-branching

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

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 $s = append(h, t)$

Branching fragment search(s:List,x:Element) has
specification:
    result defined by cases:
        if $x \in \operatorname{ran} s$ then report found
            with output i:Nat such that $s(i) = x$
        else report notfound
```

Figure 4: Specifications of some branching fragments on lists.

nodes of the tree correspond to bindings to local variables of the values returned by simple fragment calls or variables. Branching nodes correspond to calls to branching fragments, labelled by the corresponding reports; where branches return values, these values are bound to local variables. The leaves of the tree define the fragment's output values. An *abort* statement is provided for use in branches which will never be executed. Fig. 5 gives some examples.

Recursive calls and mutual recursion are allowed, provided the recursion eventually terminates. To establish termination, the CARE user supplies a variant function (or *variant* for short) whose value decreases on recursive calls. For the purposes of this introductory paper, a variant will be an $\mathbb{N}$-valued function defined on the input variables; in the full CARE method, however the variant consists of a measure and a well-founded ordering.

The bodies of higher-level branching fragments are similar in form to those of simple fragments, except that a report is produced at non-aborting leaves, together with output values, if appropriate (see e.g. Fig. 6).

5 Fragment verification

The purpose of fragment verification is to check that a fragment's implementation satisfies its specification. To a large extent, fragments are verified indi-

```
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: $\#u$.
```

Figure 5: Part of a Care program for reversing 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 $s = append(h, t)$.
implementation:
    case null(s) of
      yes:  report empty.
      no:   report nonempty and return car(s),cdr(s).
```

Figure 6: A fragment for decomposing lists.

vidually, which leads to an incremental style of working. This section outlines an informal semantics for fragments and describes how to reason about their correctness.

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

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

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

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

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

If all of the proof obligations can be shown to be logical consequences of the theory of the problem domain, the fragment set is guaranteed to be correct, in the sense that execution of a fragment on input values which satisfy its precondition will terminate and return a result which satisfies the fragment's specified postcondition.

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 postcondition. The proof obligations are illustrated on examples below. The interested reader is referred to [14] for a more detailed treatment of proof obligation generation and its justification.

5.1 Partial correctness

For simple fragments, each (non-aborting) leaf must satisfy the postcondition given in the fragment's specification. In proving the postcondition of a given leaf, the precondition of the fragment can be assumed to hold, as can the postconditions of fragment calls made along the path after making suitable variable substitutions. For any branching fragment calls in the path, the appropriate guard of the corresponding fragment specification can be assumed.

The structure of proof obligations follows exactly the structure of (paths through) the fragment implementation. For example, the fragment `reverse` given in Fig. 5 has the following partial correctness proof obligation:

$$\begin{array}{l}\forall\, s : \mathrm{seq}\, \mathit{Elem} \bullet \\ \quad \forall\, u, v : \mathrm{seq}\, \mathit{Elem} \bullet u = s \land v = \langle\,\rangle \Rightarrow \\ \quad\quad \forall\, r : \mathrm{seq}\, \mathit{Elem} \bullet r = \mathit{rev}(u) \frown v \Rightarrow \\ \quad\quad\quad r = \mathit{rev}(s)\end{array}$$

The prototype toolset includes a simplifier which reduces the above formula to:

$$\forall\, s : \mathrm{seq}\, \mathit{Elem} \bullet \mathit{rev}(s) \frown \langle\,\rangle = \mathit{rev}(s)$$

which is provable from the theory of lists. Similarly the partial correctness proof obligation for the second leaf of `revAcc` is:

$$\begin{array}{l}\forall\, u, v : \mathrm{seq}\, \mathit{Elem} \bullet \neg\, (\#u = 0) \Rightarrow \\ \quad \forall\, h : \mathit{Elem};\ t : \mathrm{seq}\, \mathit{Elem} \bullet u = \mathit{append}(h, t) \Rightarrow \\ \quad\quad \mathit{rev}(t) \frown \mathit{append}(h, v) = \mathit{rev}(u) \frown v\end{array}$$

To prove partial correctness for a branching fragment, it is necessary to show that the appropriate guard and postcondition (if any) hold. Consider for example the second leaf in the implementation tree for `decomposeList` given in Fig. 6. The report at this leaf is `nonempty` so the full proof is:

$$\begin{array}{l}\forall\, s : \mathrm{seq}\, \mathit{Elem} \bullet \neg\, (\#s = 0) \Rightarrow \\ \quad \forall\, h : \mathit{Elem} \bullet h = \mathit{head}\, s \Rightarrow \\ \quad\quad \forall\, t : \mathrm{seq}\, \mathit{Elem} \bullet t = \mathit{tail}\, s \Rightarrow \\ \quad\quad\quad \neg\, (\#s = 0) \land s = \mathit{append}(h, t)\end{array}$$

5.2 Termination

Termination proof obligations are generated for each minimal loop in the call graph for a Care program [14]. The termination proof obligation for the recursive call in the `revAcc` fragment is:

$$\begin{array}{l}\forall\, u, v : \mathrm{seq}\, \mathit{Elem} \bullet \neg\, (\#u = 0) \Rightarrow \\ \quad \forall\, h : \mathit{Elem};\ t : \mathrm{seq}\, \mathit{Elem} \bullet u = \mathit{append}(h, t) \Rightarrow 0 \leq \#t < \#u\end{array}$$

5.3 Well-formedness

Well-formedness proof obligations are generated for each call to a fragment with a nontrivial precondition. To illustrate the well-formedness proof obligation consider the fragment `end` given in Fig 7.

In this implementation, the following fragment calls have nontrivial preconditions: `cdr(s)`, `car(s)`, `end(t)`. Well-formedness of the first two calls follows

```
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: $\#s$.
```

Figure 7: Calculating the last element of a list.

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. This is given by the following proof obligation:

$$\begin{array}{l}\forall\, s : \operatorname{seq} Elem \bullet \#s \neq 0 \Rightarrow \\ \quad \forall\, t : \operatorname{seq} Elem \bullet t = tail\; s \Rightarrow \\ \qquad \neg\,(\#t = 0) \Rightarrow \#t \neq 0\end{array}$$

5.4 Non-execution

Finally, as an example of the non-execution proof obligation, consider the second leaf in the implementation tree for `cadr(s)` in Fig. 8. The non-execution proof obligation for the second leaf is given by:

$$\begin{array}{l}\forall\, s : \operatorname{seq} Elem \bullet \#s \geq 2 \Rightarrow \\ \quad \forall\, a : Elem;\; u : \operatorname{seq} Elem \bullet s = append(a, u) \Rightarrow \\ \qquad \#u = 0 \Rightarrow \text{false}\end{array}$$

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

6 Templates

6.1 Overview

A *template* is a reusable, parameterised collection of CARE types, fragments and theories, which together encapsulate a piece of design knowledge. Templates

```
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.
```

Figure 8: Calculating the second element of a list.

include commonly used algorithms and data refinements, collections of primitive components, and specific application domain theories.

A template consists of an identifier which uniquely identifies the template together with one or more of the following:

formal parameters which stand for mathematical sets, functions or relations. A signature is also given for function and relation parameters.

applicability conditions which are sufficient to verify the template's correctness.

definitions of any new mathematical constructs required in the template.

fragments and types which may be either implemented or specified-only. Implemented components can in turn either be primitive or higher-level.

The instantiation and use of templates is explained below. The names of types and fragments used within a template can be renamed by the user to suit their application without changing the meaning of template (we refer to such names as "textual parameters").

Applicability conditions dictate the circumstances under which the template can be used. It is envisaged that the template's correctness is established once, off-line by a CARE expert, the applicability conditions providing sufficient conditions to establish the template's correctness. In this way, the software engineer's verification task is reduced from proving the whole fragment set to showing that the template's applicability conditions are satisfied, which is generally a much simpler task.

6.2 A template for accumulators

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: Fig. 9 defines a template which encapsulates the accumulator strategy described above. The template is parameterised by the kinds of elements of the list ($Elem$), the type of the accumulated value (Acc), and the function (f) which describes the overall result of processing the list. Three auxiliary parameters ($base$, hd, dh) also need to be instantiated by the user: their purpose is explained below.

The template introduces and defines a function $fold$ which takes an accumulated value and a list, and returns a new accumulated value by applying hd successively to each of the elements of the list, starting from the left. In Section 6.5 below we prove that the applicability conditions are sufficient to establish correctness of the template. The proof involves a double induction, 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.

As a special case, note that if $Acc = Elem$ and dh is associative and commutative (AC), then applicability conditions 3 and 4 are automatically true upon defining $hd(a, x)$ to be $dh(x, a)$. Keith Harwood has developed a set of accumulator templates with a whole range of applicability conditions [12].

6.3 Applying the template

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

> Type `Nat` has specification: $\mathbb{N}$
> Type `NatList` has specification: $\operatorname{seq}\mathbb{N}$
>
> Fragment `sum(s:NatList)` has
> specification: output `n:Nat` such that $n = sum(s)$.

where

> Theory definition of sum.
> $sum : \operatorname{seq}\mathbb{Z} \rightarrow \mathbb{Z}$;
> $sum\langle\,\rangle = 0$,
> $\forall\, x : \mathbb{Z} \bullet sum\langle x\rangle = x$,
> $\forall\, s, t : \operatorname{seq}\mathbb{Z} \bullet sum(s \frown t) = sum(s) + sum(t)$.

The user would match these components with the type components in the template and `processList` by supplying the following partial instantiation:

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

```
Template Accumulator is
    Formal parameters:
        $Elem$    $f : \mathrm{seq}\, Elem \rightarrow Acc$      $hd : Acc \times Elem \rightarrow Acc$
        $Acc$     $base : Acc$                                   $dh : Elem \times Acc \rightarrow Acc$
    Applicability conditions:
            1. $f\langle\,\rangle = base$,
            2. $\forall\, h : Elem;\ t : \mathrm{seq}\, Elem \bullet f(append(h,t)) = dh(h, f(t))$,
            3. $\forall\, x : Elem \bullet hd(base, x) = dh(x, base)$,
            4. $\forall\, x, y : Elem;\ a : Acc \bullet hd(dh(x,a), y) = dh(x, hd(a,y))$.
    Type Elem has specification: $Elem$
    Type Acc has specification: $Acc$
    Type List has specification: $\mathrm{seq}\, Elem$

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

    Fragment accumulator(s:List,a:Acc) has
    specification: output b:Acc such that $b = fold(a, s)$.
    implementation:
        case decomposeList(s) of
          empty:     return a.
          nonempty:  assign outputs to h:Elem,t:List;
                     return accumulator(t,processElem(a,h)).
    variant: $\#s$.

    Theory definition of $fold$.
        $fold : Acc \times \mathrm{seq}\, Elem \rightarrow Acc$;
        $\forall\, a : Acc \bullet fold(a, \langle\,\rangle) = a$,
        $\forall\, a : Acc;\ h : Elem;\ t : \mathrm{seq}\, Elem \bullet fold(a, append(h,t)) = fold(hd(a,h), t)$.

    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 decomposeList(s:List) has specification:
        result defined by cases:
            if $\#s = 0$ then report empty
            else report nonempty
                with outputs h:Elem,t:List such that $s = append(h,t)$.

end template.
```

Figure 9: A template for list accumulators

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

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

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

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

The algorithm can be completed by implementing `base` and `processElem` by fragments `zero` and `addition` respectively.

6.4 Examples

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:Nat` 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

$$largerOf : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z};$$
$$\forall x, y : \mathbb{Z} \bullet \text{if } x \leq y \text{ then } largerOf(x, y) = y \text{ else } largerOf(x, y) = x$$

The applicability conditions are easily shown to hold in this case.

As a non-AC example, consider the following fragment specification:

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

Type `List` has specification: seq $Elem$.

Matching with the template and solving the first two applicability conditions gives:

$$Elem \leadsto Elem, \quad Acc \leadsto \text{seq}\, Elem, \quad f(s) \leadsto rev(s), \quad base \leadsto \langle\,\rangle, \quad dh(x, a) \leadsto a \frown \langle x \rangle$$

Applicability conditions 3 and 4 become

$$hd(\langle\,\rangle, x) = \langle x \rangle$$
$$hd(a \frown \langle x \rangle, y) = hd(a, y) \frown \langle x \rangle$$

which can be solved by defining $hd(a, x) \leadsto \langle x \rangle \frown a$. The program can be completed by renaming `base` to `nil` and defining

> Fragment `processElem(b:List,x:Element)` has implementation: `cons(x,b)`.

The reader will see that the resulting program is just a variant of the one given in Fig. 5.

6.5 Verification of the template

To verify the template it is necessary to show that the definition of *fold* is mathematically consistent and that the implementations of `processList` and `accumulator` satisfy their specifications.

The well-definedness of *fold* follows by list induction over its second argument. Details are left to the reader.

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

$$\forall\, s : \mathrm{seq}\, Elem \bullet \# s = 0 \Rightarrow fold(a, s) = a$$
$$\forall\, s : \mathrm{seq}\, Elem \bullet s = append(h, t) \Rightarrow fold(a, s) = fold(hd(a, h), t)$$

Termination of `accumulator` follows from the fact that

$$s = append(h, t) \Rightarrow 0 \leq \# t < \# s$$

The partial correctness proof obligation for `processList` is

$$\forall\, s : \mathrm{seq}\, Elem \bullet f(s) = fold(base, s). \qquad (1)$$

Before proving this proof obligation we shall establish three useful lemmas:

Lemma 6.1

$$\forall\, a : Acc;\ e : Elem \bullet fold(a, \langle e \rangle) = hd(a, e)$$

The proof follows easily form the definition of *fold*.

Lemma 6.2

$$\forall\, a : Acc;\ e : Elem;\ u : \mathrm{seq}\, Elem \bullet fold(a, u \frown \langle e \rangle) = hd(fold(a, u), e).$$

Proof. This can be established by (left) induction on the list u. The base case is:

$$fold(a, \langle\rangle \frown \langle e\rangle) = hd(fold(a, \langle\rangle), e)$$

which can be easily shown using the definition of *fold*. Now assuming that the hypothesis holds for $u = t$, we are required to prove that it also holds for $u = \langle h\rangle \frown t$; that is we want to show that:

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

Focusing on the left-hand side of the above equation:

$$\begin{array}{lll} fold(a, \langle h\rangle \frown t \frown \langle e\rangle) & & \\ \quad = & fold(hd(a, h), t \frown \langle e\rangle) & \text{by definition of } fold \\ \quad = & hd(fold(hd(a, h), t), e) & \text{by induction hypothesis} \\ \quad = & hd(fold(a, \langle h\rangle \frown t), e) & \text{by definition of } fold \end{array}$$

□

Lemma 6.3

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

Proof. This can be established by (right) induction on the list t. The base case where $t = \langle\rangle$ is:

$$dh(h, fold(base, \langle\rangle)) = fold(base, \langle h\rangle)$$

Focusing on the left hand side:

$$\begin{array}{lll} dh(h, fold(base, \langle\rangle)) & & \\ \quad = & dh(h, base) & \text{by definition of } fold \\ \quad = & hd(base, h) & \text{by applicability condition 3} \\ \quad = & fold(base, \langle h\rangle) & \text{by lemma 6.1} \end{array}$$

Now consider the inductive step, where we assume the equation is true for $t = u$ and required to prove for $t = u \frown \langle e\rangle$, that is we want to show:

$$dh(h, fold(base, u \frown \langle e\rangle)) = fold(base, \langle h\rangle \frown u \frown \langle e\rangle)$$

Focusing on the left-hand side of the above equation:

$$\begin{array}{lll} dh(h, fold(base, u \frown \langle e\rangle)) & & \\ \quad = & dh(h, hd(fold(base, u), e)) & \text{by lemma 6.2} \\ \quad = & hd(dh(h, fold(base, u)), e) & \text{by applicability condition 4} \\ \quad = & hd(fold(base, \langle h\rangle \frown u), e) & \text{by induction hypothesis} \\ \quad = & fold(base, (\langle h\rangle \frown t) \frown \langle u\rangle) & \text{by lemma 6.2} \\ \quad = & fold(base, \langle h\rangle \frown (t \frown \langle u\rangle)) & \text{as required} \end{array}$$

□

Having established these lemmas, we can now return to proving the partial correctness proof obligation for `processList`, given by equation (1). This can which can be established by (left) induction on s. We begin by considering the base case:

$$f(\langle\,\rangle) = fold(base, \langle\,\rangle)$$

which follows immediately from first applicability condition and the definition of $fold$.

The induction step amounts to proving

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

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

$$\begin{array}{lll} f(\langle h\rangle \frown t) & & \\ \quad = & dh(h, f(t)) & \text{by applicability condition 2} \\ \quad = & dh(h, fold(base, t)) & \text{by induction hypothesis} \\ \quad = & fold(base, \langle h\rangle \frown t) & \text{by lemma 6.3} \end{array}$$

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

7 Comparison with related projects

The Care approach to formal program verification is broadly similar to a number of model-oriented, hierarchical development methods, such as VDM [16] and B [18], in which abstract high-level specifications are progressively transformed into low-level executable specifications through a stepwise process of adding algorithmic detail and refining data representations. In style, it is perhaps closer to the Refinement Calculus [23], in as-much-as it supports incremental (fine-grained) development steps. In fact, Care is “expressively equivalent” to the Refinement Calculus in the sense that rules from the latter can easily be expressed as Care templates and vice-versa [21]. 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.

To the best of our knowledge, no comparable tool-set is available for VDM or Z. The `mural` environment [15] 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) but none support general refinement, with the notable exception of `Cogito` [5], 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 [4] 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.

KIDS [26] 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 [17] 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 [28], 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.

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.

8 Conclusions

This paper has outlined the CARE approach to constructing and verifying software. The approach has been developed in response to identified industrial needs for a formal development method which does not require the user to be an expert in mathematical logic. The key features of the CARE approach are:

1. an integrated notation for specification and implementation;

2. support for incremental refinement, from abstract specification through to source code in common programming languages, with integrated checks that the program is meeting its specification; and

3. use of pre-verified templates for common design steps.

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 performed off-line or consigned to automated tools, allowing the software engineer to concentrate instead on the design and development of useable, efficient pieces of software.

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. It can even be used with programming languages which do not have a full formal semantics.

The Care method has been trialled at Telectronics in a five-day intensive training course attended by senior software engineers not directly involved in the Care project. It is testimony to the effectiveness of the method and toolset that it can be used after so little training (albeit for fairly small examples). Telectronics plan to try the approach to develop part of the software for their next product range.

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.

References

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

[2] H. Barringer, J.H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[3] 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.

[4] 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.

[5] 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.

[6] B. Carré, J. Garnsworthy, and W. Marsh. SPARK: a safety-related Ada subset. In *Proc. 1992 Ada UK Conference*, London Docklands, 1992.

[7] D. Craigen et al. EVES: an overview. In S. Prehn and W.J. Toetenel, editors, *Proceedings of VDM'91*. Springer-Verlag, 1991.

[8] M.-C. Gaudel and J. Woodcock, editors. *Proc. 3rd Int. Symp. of Formal Methods Europe (FME'96): Industrial Benefit and Advances in Formal Methods*. Springer Verlag, March 1996. Lecture Notes in Comp. Sci. Vol. 1051.

[9] 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.

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

[11] K. Harwood. Towards tools for formal correctness. In *The Fifth Australian Software Engineering Conference*, pages 153–158. IREE Australia, May 1990.

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

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

[14] 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.

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

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

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

[18] K. Lano. *The B Language and Method: A Guide to Practical Formal Development.* FACIT Series. Springer-Verlag, 1996.

[19] 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.

[20] 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.

[21] P. Lindsay. Expressing program developments from the refinement calculus in care. In *Proc. 4th Australian Refinement Workshop*, Sydney, Australia, 1995. Also appears as SVRC TR 94-10.

[22] P.A. Lindsay. The data logger case study in CARE. In *Proc 5th Australasian Refinement Workshop (ARW'96)*, 1996. Also appears as SVRC TR 95-10.

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

[24] M. Naftalin, T. Denvir, and M. Bertran, editors. *Proc. 2nd Int. Symp. of Formal Methods Europe (FME'94): Industrial Benefits of Formal Methods.* Springer Verlag, October 1994. Lecture Notes in Comp. Sci. Vol. 873.

[25] J.E. Nicholls. Z Notation, Version 1.2. Technical report, Z Standards Panel, September 1995.

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

[27] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, New York, 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] M. Utting and K. Whitwell. `Ergo` user manual. Technical Report 93-19, Software Verification Research Centre, revised March 1994.

[30] M. Weber, M. Simons, and C. Lafontaine. *The Generic Development Language Deva*, volume 738 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.