But who will guard the guardians?

Dave Maley, Queen's University of Belfast (d.maley@stmarys-belfast.ac.uk)

Ivor Spence, Queen's University of Belfast (i.spence@qub.ac.uk)

Abstract

Design by Contract is widely accepted as a valuable software design methodology for improving software quality. Its incorporation into the Eiffel language has been largely responsible for this. However, the Eiffel language restricts the expressivity of what may be verified in the contract primarily to propositional logic. When the contract is non-trivial, if we wish to retain contract-checking then we must write our own validation routines. How can we be sure that the code that checks the correctness is itself correct? The question was first posed by Juvenal in the first century A.D. 'Sed quis custodiet ipsos custodes?'. The level of genericity now possible in C++ at last offers a step forward.

1 Introduction

Design by Contract, as supported in the Eiffel language, provides two separate facilities: the facility to express the behaviour of objects in a formal notation, and the facility to check that the stated behaviour is indeed observed during the execution of the system. Although there are many examples of behaviour that is expressible but not checkable, the Eiffel Design by Contract mechanism does not directly permit the expression of many contracts that are checkable (although user-defined checks are permitted in the form of function calls). This is because the provision of the two separate facilities within a single mechanism creates the need for a trade off between expressivity and efficient checkability. Deciding on where the cut-off point for checkability lies is described by Bertrand Meyer [1] thus:

'Determining that threshold is clearly a matter for personal judgement. I have been surprised that, for the software community at large, the threshold has not moved since the first edition of this book. Our field needs more formality, but the profession has not realised it yet.'

Propositional logic is fundamentally inadequate for describing the behaviour of container data structures; in particular, any expression regarding some or all of the elements of a container require to facility to express universal and existential quantification. It is a rare specification that does not have to deal with aggregate data, and as soon as it necessary to encode user-defined functions to verify constraints, the question arises of how to be sure that the code that checks that the constraint is satisfied is itself correct.

Our work in the scientific programming field has led us to explore the combination of VDM++ [2] and C++ [3, 4] for the expression and verification of contracts. VDM++ permits the expression of second order and some higher order expressions of the predicate calculus, so quantification, non-determinism and comprehension are all expressible in the language. The multi-paradigm design philosophy of C++, and especially its support for generic programming, facilitate the provision of sophisticated constraint checking apparatus.

The question of applying Design by Contract beyond the Eiffel context is discussed by a number of authors. As Jonkers [5] points out,

"Nowadays the pre- and postcondition technique is considered a standard technique in software development as it is being taught in almost every basic software engineering course. This gives the impression that the technique has fully matured and that it can be applied everyday in software development practice. The fact that this is not really the case is camouflaged by the sloppy and informal way pre- and postconditions are generally used in practice."

Making best use of Design by Contract is a topic of active research interest. At TOOLS Pacific in 1995, James McKim [6] put forward ten principles of class interface design (see also [7, 8, 9]), which advocated amongst other things not letting the specification be limited to what can be expressed in Eiffel.

In [10], Mitchell, Howse and Hamie discuss an approach to deriving contracts for abstract data types which they term contract-oriented specification. Abstract data types are usually specified using equational (algebraic) specifications. Such specifications do not map readily into the assertions of the pre- and postcondition technique. The authors suggest that algebraic specifications can be rewritten in a form from which the original properties can be derived as theorems (thus proving that the original properties are maintained), but in their rewritten form are amenable to systematic encoding into executable pre- and postconditions.

The behavioural interface specification language Larch/C++ [11, 12], in common with many other specification languages, provides the facility to express preconditions, postconditions, frame conditions and invariants. However, it also provides a number of additional facilities, such as partial correctness, redundancy and history constraints, for which a strong case can be made for inclusion in other specification languages [13], illustrating Jonkers' point that the approach has not yet fully matured. The emphasis is very much on the issue of expressivity, rather than the logic and formality of, for example, VDM++. One of the main uses of the language is seen as being in the documentation of existing code.

Jézéquel, Train and Mingins [14] combine design patterns with software contracts, showing that the intention behind a design can be made more explicit, so that component objects, their collaborations and the distribution of responsibilities amongst them can be made more concrete. This idea is also demonstrated in [15], which describes the first production use of the language R++. R++ extends C++ by the addition of object-oriented rules.

All of this work highlights the gap between what it is desirable to specify, and what it is possible to verify: and also, to what extent it is possible to verify that those things that have been verified have been verified correctly. In Eiffel, contracts involving anything more sophisticated than propositional logic requires user-defined functions. This is also true in C++, but the level of genericity available in C++ makes it possible to maintain a direct mapping between specification statements in VDM++ and executable predicates in C++.

In this paper we consider a straightforward and commonly used data structure from finite element analysis, the mesh. It is shown how VDM++ can be used to express the properties of the mesh, and that these properties can be verified during system execution when implemented in C++, and with a substantial degree of confidence that the checks do indeed verify the stated behaviour. The approach has been applied to specifications from three different disciplines – engineering, computational physics and genetics – and as a result we are confident that the approach will have widespread utility and will make a contribution to the improvement of software quality.

2 An Example Problem

Finite element analysis is a topic becoming increasingly popular in engineering for solving structural analysis and other problems. The solution method involves the construction of a data structure known as a mesh [16]. In this example, a simple two-dimensional mesh will be considered. A portion of a mesh is shown in Figure 1.

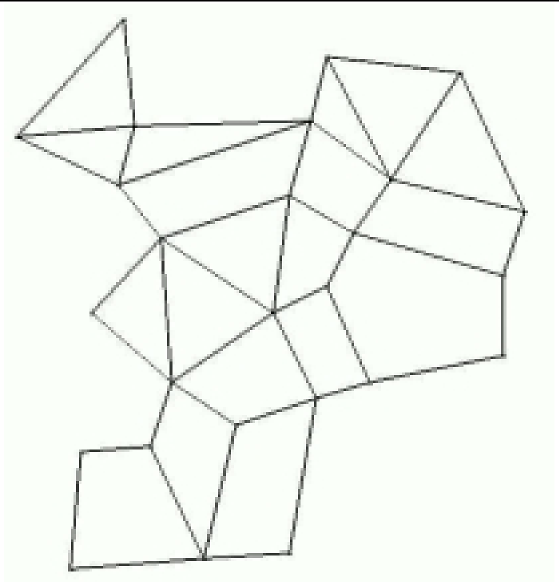


Figure 1. A mesh consisting of 26 nodes and 19 elements.

In a real situation, a mesh will be three-dimensional and consist of many tens of thousands of nodes and elements.

For a given element, there are several properties that it is useful to determine. For example, it is useful to be able to determine whether a given point is enclosed by the given element. This property is easy to state in natural language, but is not necessarily so easy to state formally. A second property of interest is being able to identify the centroid of an element. The centroid of an element is essentially its mid-point: it is calculated as the average of the coordinates of its constituent nodes. Centroids are useful for refining meshes – adding more nodes and elements – and also for smoothing – reducing the irregularities of a mesh. A further significant property (set by the user) is the maximum angle between two edges of an element.

Centroid refinement is the process of replacing each element in the mesh with several smaller elements, each of which is formed from two adjacent nodes of the original element and the centroid of the element. Thus all the elements in the resulting mesh are triangular. The effect of applying centroid refinement to the mesh in Figure 1 is shown in Figure 2.

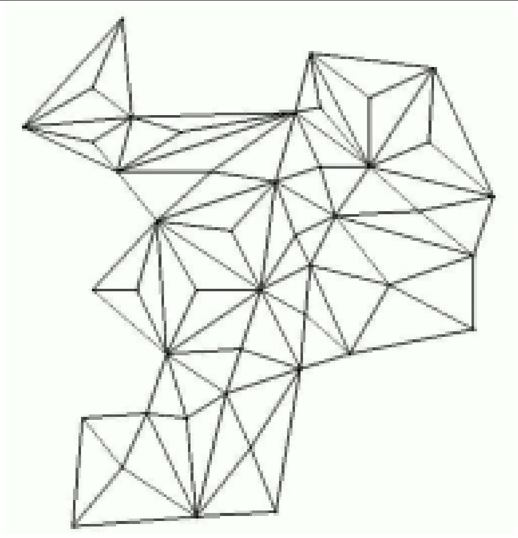


Figure 2. The mesh of Figure 1 after centroid refinement.

Centroid refinement is also used in the operation of Laplacian smoothing. In this operation, each interior node of the mesh is repositioned at the centroid of its immediate neighbours. The effect of applying the Laplacian smoothing operation to the mesh in Figure 2 is shown in Figure 3.

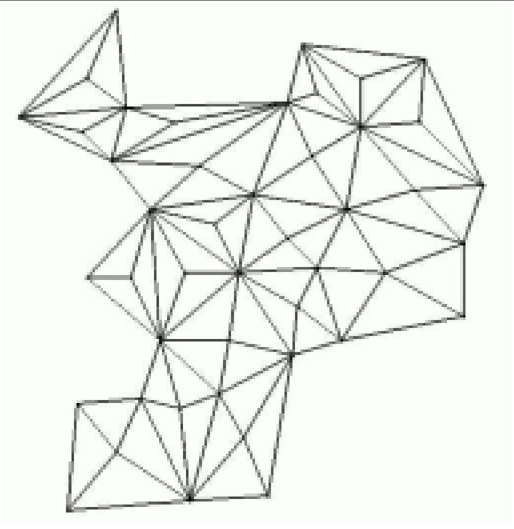


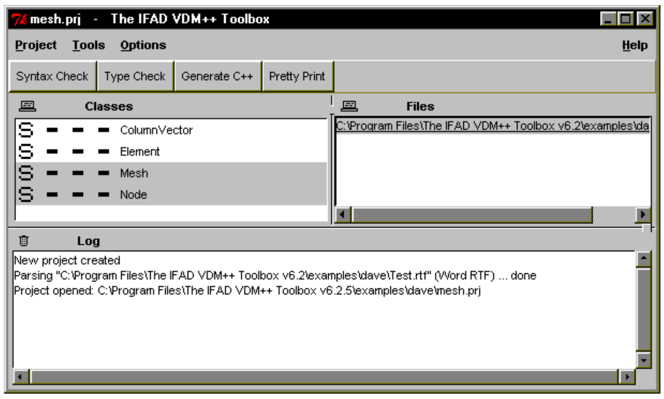
Figure 3. The mesh of Figure 2 after Laplacian smoothing.

Clearly a mesh as illustrated is a relatively simple data structure, representable for instance by a list of nodes and a list of elements. However, it would not be possible to express even the properties of the data structure, let alone its operations, in the constraint checking language of Eiffel. Consequently a more expressive language is needed if we are to state the properties of a mesh [17]. What is needed above all else are the universal and existential quantifiers ‘forall’ and ‘exists’.

3 Stating Properties Using VDM++

VDM++ is an object-oriented extension of VDM-SL, the specification language of the Vienna Development Method [18]. VDM-SL and Z [19] are often cited as the two most widely-used model-based specification languages in formal methods. A VDM++ specification consists of a set of *class* definitions. VDM++ classes are quite similar in some respects to classes in classic object-oriented languages such as C++ or SmallTalk. They provide a template for *objects*, which possess attributes and operations (*methods*) whose types and properties are defined in the class. Thus VDM++ attributes correspond to C++ *data members* and VDM++ methods to C++ *member functions*.

The IFAD toolbox [20] is a set of tools that permit the development and analysis of precise models of computing systems using VDM++. The tools can automatically check and validate these models, and range from traditional syntax and type checking tools to a powerful interpreter and debugger. Test suite facilities are also included, as is the facility to support the automatic generation of C++ code from the specification. The toolbox front end is shown in **Figure 4**, displaying the classes of the `Mesh` system.



```
C:\Program Files\The IFAD VDM++ Toolbox v6.2\examples\da
```

```
New project created
Parsing "C:\Program Files\The IFAD VDM++ Toolbox v6.2\examples\dave\Test.rtf" (Word RTF) ... done
Project opened: C:\Program Files\The IFAD VDM++ Toolbox v6.2.5\examples\dave\mesh.prj
```

Figure 4.

Screenshot of the IFAD VDM++ Toolbox

3.1 *The `Element` Class*

The three principal classes involved in the example are `Node`, `Element` and `Mesh`. Consider first the `Element` class. The `Element` class is declared to have an instance variable of type `seq of Node` (a `seq`uence is one of the fundamental data types in VDM-SL).

```
instance variables
..
nodes : seq of Node;
```

3.1.1 The invariant of `Element`

The invariant must state that

- there are at least three nodes in the sequence;
- the nodes in the sequence are distinct from one another;
- the angle between any three adjacent nodes in any element is less than a certain threshold, and that this threshold is less than π radians.

The invariant of the class is expressed in VDM++ as follows:

```
inv
len(nodes)>2 and
len(nodes)=card(elems(nodes)) and
maxAngle < pi and
forall i in set inds(nodes) &
  nodes(i).getAngle(prev(nodes(i)), next(nodes(i)))<maxAngle
```

The invariant makes use of some of the predefined operations of VDM++. The `len(<sequence>)` operator returns the number of members in a sequence. The `elems(<sequence>)` operator returns a set consisting of the members of a sequence. The `inds(<sequence>)` operator returns a set consisting of indices of the the members of a sequence. The `card(<set>)` operator returns the number of elements in a set. Since a set cannot contain duplicates, if the length of a sequence is equal to the cardinality of the set formed from its members, the members must be distinct.

VDM++ provides the quantifiers `forall`, `exists` and `exists1`. The first line of the final clause of the invariant reads "for all integers i in the set of indices of the sequence called `nodes`, …". These quantifiers can be nested to arbitrary depth. The ampersand symbol is a delimiter that can be read as "it is the case that". VDM++ also has a mathematical syntax, but since the IFAD tool can generate appropriate Latex code from plain text such as that above, there is no need to grapple with the intricacies of mathematical typesetting.

3.1.2 Class Methods of `Element`

Method `next(p : Node)` returns the `Node` following parameter `Node p` in the `Element`. Its VDM++ specification is

```
next(p : Node) q : Node
  ext rd nodes
  pre
  p in set  elems(nodes)
  post
  exists i in set inds(nodes) &
    p=nodes(i) <=> q= nodes((i+1) mod len(nodes));
```

The behaviour of the method is stated in terms of a precondition and a postcondition. (Implicit specification in this form is only one of the options available in VDM++ for stating the behaviour of operations). The precondition states that the `Node` passed to `next(..)` must be a `Node` of the `Element`.

The `Element` class provides a `prev(Node)` method complementary to `next(Node)`, a `getIndex()` method which returns the sequence index of a given `Node`, and a method `centroid()`, as described above. It also provides methods `adjacent(Node, Node)`, which indicates whether two `Node`s are adjacent in the `Element` and a `method encloses(Node)`, which indicates whether an `Element` encloses a `Node`. The specifications of these methods can be found – along with the full specification of the mesh – in WebAppendix 1.

3.2 The Mesh Class

The `Mesh` class has two instance variables, a sequence of `Nodes` and a sequence of `Elements`.

```
instance variables
..
meshNodes : seq of Node;
meshElements : seq of Element;
```

The invariant of the `Mesh` class deals largely with the consistency of the two sequences. It states that

- all the `Nodes` in `meshNodes` are distinct, and are part of at least one `Element`;
- for all `Elements` in `meshElements`, the members of the `Element` are members of meshNodes;
- `Elements` do not overlap.

3.2.1 Class Methods of `Mesh`

The two principal methods of the `Mesh` class, `centroidRefine()` and `Laplace()`, have been described earlier. The other methods of the class support the principal methods. The postcondition of the Laplacian smoothing operation is shown below. It makes use of the set difference operator \, and also the VDM equivalent of the Eiffel **`old`** operator, the postfix tilde (~).

```
Laplace()
ext wr meshNodes
post
```

```
forall p in set elems(meshNodes~) &
  interior(p) =>
  exists1 i in set inds(meshNodes) &
    elems(meshNodes)\{meshNodes(i)}=elems(meshNodes~)\{p}
    and meshNodes(i)=neighbours(p).centroid();
```

The VDM++ language permits the precise specification of the properties of the `Mesh` component. These properties can be stated in terms of an abstract model, without concern for how this behaviour can be produced algorithmically or for the details of a particular implementation language. This frees the mind from unnecessary detail, facilitating analytical thought and communication of ideas. However, Bertrand Russell's aphorism that 'the advantages of implicit definition over construction are roughly those of theft over honest toil' holds true, and the honest toil stage remains. The next section considers how best to verify that the outcome of the honest toil is the same as that obtained by theft.

4 Implementing Constraint-checking in C++

There are two distinct aspects to supporting run-time assertion monitoring in C++: providing a framework in which the properties can be monitored, and providing a framework in which the properties can be expressed. In Eiffel, the monitoring framework is built-in. In C++ it can provided either by library adaptation (in the case of STL containers) [21], or else in the general case (including subtyping) by the use of generic function objects [22]. The instantiation of generic function objects permits the declaration of method preconditions and postconditions in which the invariant of the class is automatically verified on entry to and exit from the method. Also, the return value of the method (if it has one) and the '**`old`**' value of the object are automatically made available to the postcondition, all without altering the unpredicated code of the method. The necessary support structure is created simply by declaring instantiations of templates provided in a library, and the preconditions and postconditions are defined as specialisations, so the support architecture appears in practice as if it were built into the language. An example of such a template is shown in Appendix I.

This paper considers the second issue, providing a framework in which the properties can be expressed. The aim is simple: to reduce the gap between the two expressions of the properties, the statement of the behaviour given in the VDM++ and the verification of the behaviour given in the C++.

4.1 The STL

In order to understand the material that follows it is necessary to be familiar with the STL concepts of iterators and function objects (and the STL in general). It is also necessary to be familiar with C++'s concept of a `typedef`. A `typedef` introduces a new name for a type. When the old type is an instantiation involving several parameters, `typedefs` can reduce verbosity and also add meaning. `Typedefs` within class scopes facilitate the level of genericity possible in some STL algorithms, providing access to the type structure of the instantiating class – in some cases, the algorithms are decoupled from the containers they operate on to the extent that the requirements on the container are simply that it has the appropriate type names in scope.

The STL algorithm `find_if(..)` illustrates the use of iterators and function objects.

```
template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

It might be called as follows:

```
void f(vector<int>& v1)
{
  find_if(v.begin(), v.end(), pred());
}
```

It is used to search a container to find an element that satisfies a given predicate. The type of container need not be known: it simply is required to implement the iterator protocol, so that dereferencing the iterator (“`*first`”) and prefix incrementing it (“`++first`”) have the expected effect. The predicate is a function object: an object for which the function application operator `operator()(..)` is defined. Thus “`pred(*first)`” means “apply the `operator()(..)` method to the `pred` object with parameter `*first`”. Clearly the function application operator of class `pred` must therefore be defined to return a `bool`, (or a type convertible to a `bool`). An iterator to the matching container element is returned. If the search is unsuccessful then an iterator to `last`, the element after the final element of the container, is returned.

The important point is the decoupling involved: the specific functionality of the particular search operation is encapsulated within the function object; the algorithmic steps, which would be common to all searches of that type, is factored out into the generic algorithm.

4.2 Going Beyond Propositional Logic

It is nothing new to point out the limitations of the propositional logic that adds checkability to Eiffel. Whenever extensions are considered [23, 24], it is invariably quantification that is addressed first, and this is taken as evidence of its importance.

This section explores how the ideas behind the STL can be used to bring into closer alignment the statement of the behaviour given in the VDM++ and the verification of the behaviour given in the C++. It should be reiterated that this proposal begins to tackle a problem ignored by practically all previous work: how to be sure that the code that verifies correctness is itself correct.

Consider again the invariant of the `Element` class.

```
inv
len(nodes)>2 and
len(nodes)=card(elems(nodes)) and
maxAngle < pi and
forall i in set inds(nodes) &
  nodes(i).getAngle(prev(nodes(i)), next(nodes(i)))<maxAngle
```

Clearly the first and third clauses can be encoded quite simply: these are the type of clauses that can also be expressed quite simply in the Eiffel assertion language. However, the second and fourth clauses are not so straightforward. In Eiffel they would require the user to write a function to verify the property. This is also the case in C++, but there is a significant difference: the genericity permitted by C++ makes it possible to support writing clauses using constructs that closely parallel the predicate calculus used in VDM++.

Before this is considered, it is important to mention another C++ feature – declaration of `const`-ness. As soon as user-defined verification checks are permitted, there is the danger of “letting the imperative fox back into the applicative chicken coop” [25]. In other words, the checks might actually modify the behaviour instead of simply monitoring it. This can be guarded against by declaring all such routines ‘`const`’, requesting the compiler to ensure that they have no effect on the objects whose behaviour they monitor. `const`-ness can be circumvented by casting, but that would be folly on the part of the user.

4.2.1 Writing generic forall(..) and exists(..)

Considering the second clause of the invariant of class `Element`, clearly an algorithm to create a set from the members of a sequence is required, and it is straightforward to write (Appendix II). The interesting case is the fourth clause. Verification of universal quantification over a finite set can be abstracted by the following generic algorithm.

```
template <typename In, typename UnaryOp>
bool forall(In first, In last, UnaryOp op)
{
  while (first!=last)
    if (!op(*first++))
      return false;
  return true;
}
```

This algorithm applies the function application operator of the class `UnaryOp` to each of the elements referenced through the iterators `first` and `last`. Therefore if `UnaryOp` is defined in such a way as to verify a given property, the `forall` algorithm can verify universal quantification of that property over a finite set.

The `exists` algorithm is so similar that the two algorithms can in fact be written as instantiations of the same generic algorithm `quantifier`, which takes an extra `bool` parameter which is used instead of the explicit `true` and `false`. At the same time, an additional reduction in the wordiness of the implementation can be achieved simply by recognising that it is not necessary to pass any iterators to the quantifier: a type instantiation to the type of the container suffices. Since the protocol for iterating through containers is uniform, and by definition the limits of the quantification are known, there is no need to require `first` and `last` as parameters to `forall` or `exists`. These new algorithms are shown in Appendix III.

The definition of `exists` makes use of the function adaptor `unary_negate<UnaryOp>`, which is provided as part of the STL. This function adaptor is defined so that the action of its function application operator is to return the logical complement of the function application operator of the class with which it is instantiated.

It is important to note that the definition has been amended so that the function application operator of the class `UnaryOp` will be passed an iterator as a parameter, rather than the element of the container. This is important – the properties expressed in the VDM++ make reference to the indices of the container, and the iterators provide the equivalent construct in C++. The elements themselves can of course be accessed simply by dereferencing the iterator.

4.2.2 Providing Instantiatable Function Objects

The next step is to provide a template for the class `UnaryOp`. It is fundamental to the utilisation of constrained genericity that the instantiating type may be required to display certain properties. The instantiator can be helped by providing a template class displaying some or all of the required properties, which can be specialised or inherited. The class `unary_predicate<Object, T, int>` is provided, which itself inherits from the template class `unary_function<T, bool>`, which is provided by the STL.

```
template <typename Object, typename T, int instantiationNumber=0>
struct unary_predicate : public unary_function<T, bool> {
  typedef typename T::const_iterator argument_type;
  const Object& object;
  unary_predicate(const Object& c) : object(c) {;}
  bool operator()(typename T::const_iterator) const;
};
```

The parameter `instantiationNumber` is included in case it is necessary to distinguish between two instantiations which take the same type parameters. The inclusion of the line

```
typedef typename T::const_iterator argument_type;
```

is a consequence of constrained genericity: the function adaptor `unary_negate<UnaryOp>` expects the name `argument_type` to be defined in the instantiating type `UnaryOp`.

A `unary_predicate<Object, T, int>` does not simply define a function application operator. It also defines local storage for an object reference. The reason for this is the scope of quantifiers: in VDM++, the scope of a quantifier extends as far as possible to the right. Thus in an expression involving nested quantifiers, the quantified variables in outer scopes are all accessible. In order to provide this in the C++ support, it is necessary for the function objects that support the iteration to store the information in outer scopes. This can lead to naming problems, which are discussed in a later section (§4.4)

With the above definitions available, the fourth clause can be handled by specialising the function application operator of `unary_predicate<Element, vector<NodeID> >`

```
typedef unary_predicate<Element, vector<NodeID> > elementInv;

bool elementInv::operator()(vector<NodeID>::const_iterator n) const
{
  return getAngle(object(object.prev(*n)), object(*n),
object(object.next(*n)))<object.maxAngle();
}
```

The invariant can then be written as

```
bool Element::invariant() const
{
  return \
  nodeIDs.size()<2 && \
  nodeIDs.size()=elements(nodeIDs).size() && \
  maxAngle()>0 && \
  forall(nodeIDs, elementInv(*this));
}
```

which closely resembles the properties as stated in VDM++.

The quantifier algorithm can be generalised to `quantifier_if` in the same way that the STL algorithm `find` is generalised to `find_if`. This can be used in the expression of the postcondition of the Laplacian smoothing method. The VDM++ states that it is only the interior nodes of the mesh that are moved. This can be encoded by declaring a `unary_predicate<..>` whose function application operator returns whether a `Node` is interior or not. Thus the postcondition becomes

```
return forall_if(meshnodes, laplaceCheck(), interiorCheck())
```

4.3 Going Beyond Quantification

Quantification is not the only property which can be verified by the instantiation of a generic algorithm. Verification of set, map and sequence comprehension can also be encoded generically. Consider the definition of the postcondition of the method `neighbouringPairs(Node)`.

`neighbouringPairs(p : Node)` returns a sequence of `NodePairs` in which each member of each pair is a neighbour of the given `Node`, and all three in each instance are part of the same `Element`.

```
neighbouringPairs(p : Node) s : seq of NodePair
ext wr meshNodes
pre
  p in set elems(meshNodes)
post
  let n : Element be st n=neighbours(p) in
s = [ mk_NodePair(n.meshNodes(i), n.meshNodes((i+1) mod len(n.meshNodes))) |
      i in set inds n.meshNodes]
```

The postcondition uses sequence comprehension to state the necessary behavioural property. Sequence comprehension defines a sequence by stating a set of indices, an expression that generates a sequence member for each given index, and optionally a predicate guarding inclusion in the sequence. This can be paralleled in C++ by instantiating the generic `seqComp(..)` algorithm.

```
bool NeighbouringPairs::post(const Node& n) const
{
  return seqComp(NodePairGenerator(*_object),
    getIndices(_mutableObject->neighbours(n).nodeIDs),
    result(), truePredicate<const NodeID*>());
}
```

Consideration of Appendix IV will show the complexity of nesting function objects to define `seqComp(..)`. This gives rise to a trade-off. Current wisdom indicates that the price is worth paying if the complexity is contained within the library, as with `seqComp(..)`, but may not be if it is in the client program, as in `Mesh::inv()` below.

4.4 Limitations

The advanced level of generic expression possible in C++ enables the constructions described in the two previous sections to be exploited. However, there are still limitations to what can be achieved in terms of maintaining congruence between the specification language constructs and the encoded predicates.

The invariant of the `Mesh` class will now be considered, which is where the limitations first become apparent. In VDM++, the invariant is expressed as follows:

```
inv
card set elems meshNodes = len meshNodes and
forall i in set inds(meshNodes) &
  (xists j in set inds(meshElements) &
    meshNodes(i) in set elems(meshElements(j).meshNodes) and
forall i in set inds(meshElements) &
  forall j in set inds(meshElements(i).meshNodes) &
     meshElements(i).meshNodes(j) in set elems(meshNodes) and
forall i in set inds(meshElements) &
  forall n in set allOtherNodes(i) &
    not meshElements(i).encloses(n)
```

In traditional approaches, this might be encoded as a verification routine as shown in Appendix V. It is an intricate piece of code, involving layering and nesting of loops, even though the data structure involved is quite simple. How can we be sure that this routine for checking stated behaviour performs a correct check of that behaviour? The expression in VDM++ is more comprehensible since it is not cluttered by implementation details, and because the traditional C++ routine must repeatedly code the same algorithmic structure for the verification of the properties of universal and existential quantification.

C++'s support for generic programming enables the routine to be encoded as shown below. A series of `typedef`s declares the clauses and subclauses of the invariant predicate. (The associated specialisations are shown in Appendix VI).

```
typedef unary_predicate<Mesh, vector<NodeID> > meshInv1;
typedef unary_predicate<Mesh, vector<Element> > meshInv2;
typedef unary_predicate<Mesh, vector<Node> > meshInv3;
typedef unary_predicate<Mesh, vector<Element>, 1> meshInv4;
typedef binary_predicate<Mesh, vector<Node>, vector<Node> > meshInv5;

bool Mesh::invariant() const
{
  return forall(meshElements, meshInv2(*this)) && \
         forall(meshNodes, meshInv3(*this));
}
```

However, although this approach considerably narrows the gap between the VDM++ and the C++, there remain problems. The first problem is essentially one of naming. Each logical clause and subclause that uses a quantifier – whilst not actually requiring the introduction of a new name – does require the specialisation of a function object, which will inevitably be referred to at least twice, and which if not `typedef`-ed will be difficult to read. Something similar to Eiffel's tags appears wishful thinking. Therefore unless the user is confident with the techniques for the definition and use of function objects, it is necessary to devise a scheme which can be employed consistently and with which the user can quickly attain familiarity. The best approach to this is still under investigation.

The second problem is that of scope, and again names are the issue. In VDM++ the scope of all quantifiers extends to the end of the expression. The iterators of the C++ generic quantifier predicates are not, by comparison, accessible within the definitions of the function objects they act upon unless the iterator (or its target) is passed as a parameter to the function object constructor. This can be done, and indeed can be seen to have been done in the definition of the sequence comprehension generic algorithm in Appendix IV. However, the problem arises that passing the iterator or its target into the function object scope as a parameter renames the entity in question to the formal name of the parameter, and semantically significant information is lost. This drawback can be overcome by local renaming, but this requirement falls upon the writer of the function object specialisation, which is unfortunate.

5 Conclusion

Computer software is complex, and complexity is managed by subdivision. Part of the process of subdivision involves dividing software development into stages. One price to be paid is the obligation incurred to ensure that overall integrity is maintained across the stages. Specification and implementation are fundamentally distinct stages. Therefore it is necessary to ensure that they are maintained in correct correspondence.

The Eiffel approach is to permit statements within the implementation code that state properties of the software: they are part of its specification. At run-time, the truth of these propositions can be evaluated, with failed propositions indicating that the behaviour of the implementation differs from that stated in the specification (the truth of all evaluated propositions does not of course prove the implementation to be correct with respect to the specification). The Eiffel approach is infinitely better than nothing, but it is not enough. The level of genericity in C++ enables the approach to be taken a stage further.

C++ offers the potential to create a library which supports the encoding of predicates which specify component behaviour stated in a language such as VDM++. VDM++ is not put forward here as the ideal object-oriented specification language: notably it currently lacks the power to express class genericity, parameterised object construction and function overloading; but its expressive power is far in excess of the propositional logic supported by Eiffel, and we have found it to be sufficient for our purposes. Again, the combination of VDM++ and C++ is not necessarily ideal, but it overcomes the major limitation of the Eiffel approach.

The design decision to incorporate the facility to state properties of a software component within the text of an Eiffel program is to be applauded. One of the major criticisms levelled at the use of formal specification is that the specification can sometimes get out of sync with the implementation, especially during maintenance. The Eiffel approach goes some way to reducing the likelihood of this problem occurring by integrating the two texts. The IFAD tool for syntax- and type-checking VDM++ recognises text formatted in a style tagged 'VDM++' in RTF documents, thus enabling VDM++ and natural language narrative to be mixed; it would be helpful if commercial C++ compilers such as VC++ could do the same for C++, permitting the statement of properties and their implementation to be maintained in the same document.

The problem of the specification getting out of sync with the implementation may be less likely to occur in our own field of computational science, where the formal specification describes an unchanging physical reality which is more often than not enshrined in established mathematical theory. If this part of the specification is out of step with the implementation, it is the implementation that is wrong. However, the properties of the data structures used to model the physical reality – and independent of that physical reality – are also part of the specification. It is therefore just as important in our own field that generic libraries are provided not only in implementation form but also in specification form. Again the ELKS library has the edge on comparable libraries such as the C++ STL since the intended behavioural properties of the code are stated within the text of the code. However, it is noted in [17] that this annotation is not as thorough as it might be. This is not simply because the propositional logic is insufficient to express some properties: there are expressible properties that have been overlooked.

If the industry is to increase the degree of formality as Meyer recommends, it should surely start with the libraries that are in common use. Certainly any international standardisation effort in terms of a generic algorithms and data structures library must begin with specification if only because of the need in such an undertaking for implementation language independence.

To conclude, the combination of VDM++ and C++ permits predicates specifying component behaviour to be stated in an abstract mathematical language, and then encoded into checks that can be carried out during implementation execution. This paper points the way forward in how the likelihood of introducing errors during this encoding process can be minimised. It should be noted that these checks can potentially be very expensive and are intended as a development aid – they are not intended to be performed in production code (although preconditions in library routines may optionally be validated if desired). The checks contribute to the effort required to bring a greater degree of formality to software production, which we believe is a prerequisite to improving software quality.

WebAppendix 1

http://www.stmarys-belfast.ac.uk/~d.maley/research/mesh.rtf

Appendix I

Function object template for a single parameter, subclassed class method

```
template <typename ReturnType, typename Formal1, int MethodNumber=0>
class DBC1 {
  template <typename Derived, bool Top=false, typename Dummy = typename
Derived::Base>
  class DBC : public ::DBC1<ReturnType, Formal1, MethodNumber>::DBC<typename
Derived::Base, false, typename Derived::Base::Base> {
    typedef ReturnType (Derived::* BodyType)(Formal1);
    typedef typename Derived::Base BaseType;
    typedef ReturnType (BaseType::* AncestorBodyType)(Formal1);
    typedef typename ::DBC1<ReturnType, Formal1, MethodNumber>::DBC<Derived,
Top, Dummy> This;
    typedef typename ::DBC1<ReturnType, Formal1, MethodNumber>::DBC<typename
Derived::Base, false, typename Derived::Base::Base> Ancestor;
    bool pre(Formal1) const { return true; }
    void preFailure() const { throw Assertion("preFailure", __FILE__, __LINE__);
}
    bool post(Formal1) const { return true; }
    void postFailure() const { throw Assertion("postFailure", __FILE__,
__LINE__); }
    BodyType _body;
    void handleOld(const Derived*);
    const Derived* _object;
    Derived* _mutableObject;
  protected:
    bool effectivePre(Formal1 f1) const { return Ancestor::effectivePre(f1) ||
This::pre(f1); }
    bool effectivePost(Formal1 f1) const { return Ancestor::effectivePost(f1) &&
This::post(f1); }
    bool effectiveInvariant() const { return Ancestor::effectiveInvariant() &&
This::_object->invariant(); }
    const Derived* old() const { return static_cast<const
Derived*>(Ancestor::old()); }
  public:
    DBC(Derived* object, BodyType body) : _body(body), _mutableObject(object),
_object(object), Ancestor(object, static_cast<AncestorBodyType>(body)) {
handleOld(object); }
    ReturnType operator()(Formal1);
  };

  template <typename Derived>
  class DBC<Derived, false, Derived> {
    typedef ReturnType (Derived::* BodyType)(Formal1);
    typedef typename ::DBC1<ReturnType, Formal1, MethodNumber>::DBC<Derived,
false, Derived> This;
    bool pre(Formal1) const { return true; }
    void preFailure() const { throw Assertion("preFailure", __FILE__, __LINE__);
}
    bool post(Formal1) const { return true; }
    void postFailure() const { throw Assertion("postFailure", __FILE__,
__LINE__); }
    BodyType _body;
    const Derived* _object;
    Derived* _mutableObject;
  protected:
    bool effectivePre(Formal1 f1) const { return This::pre(f1); }
    bool effectivePost(Formal1 f1) const { return This::post(f1); }
    bool effectiveInvariant() const { return This::_object->invariant(); }
    const Derived* _old;
    ReturnType* _result;
  public:
    DBC(Derived* object, BodyType body) : _body(body), _mutableObject(object),
_object(object), _old(NULL), _result(NULL) { }
    ~DBC() { if (_old) delete _old; if (_result) delete _result; }
    const Derived* const old() const { return _old; }
    ReturnType const result() const { return *_result; }
    ReturnType operator()(Formal1);
  };
};

template <typename ReturnType, typename Formal1, int MethodNumber>
  template <typename Derived, bool Top, typename Dummy>
    void DBC1<ReturnType, Formal1, MethodNumber>::DBC<Derived, Top,
Dummy>::handleOld(const Derived* dp)
    {
      if (Top) {
        if (_old)
          delete _old;
        _old = new Derived(*dp);
      }
    }

template <typename ReturnType, typename Formal1, int MethodNumber>
  template <typename Derived, bool Top, typename Dummy>
    ReturnType DBC1<ReturnType, Formal1, MethodNumber>::DBC<Derived, Top,
Dummy>::operator()(Formal1 f1)
    {
      if (!effectivePre(f1) || !effectiveInvariant())
        preFailure();
      handleOld(_object);
      _result = new ReturnType((_mutableObject->*_body)(f1));
      if (!effectivePost(f1) || !effectiveInvariant())
        postFailure();
      return *_result;
    }

template <typename ReturnType, typename Formal1, int MethodNumber>
  template <typename Derived>
    ReturnType DBC1<ReturnType, Formal1, MethodNumber>::DBC<Derived, false,
Derived>::operator()(Formal1 f1)
    {
      if (!effectivePre(f1) || !effectiveInvariant())
        preFailure();
      _result = new ReturnType((_mutableObject->*_body)(f1));
      if (!effectivePost(f1) || !effectiveInvariant())
        postFailure();
      return *_result;
    }
```

Appendix II
The elems generic algorithm.

```
template <class Container>
set<Container::value_type> elements(Container c) {
  set<Container::value_type> theSet;
  typename Container::const_iterator first=c.begin();
  while (first!=c.end()) {
    if(find(theSet.begin(), theSet.end(), *first)==theSet.end())
      theSet.insert(*first);
    ++first;
  }
  return theSet;
}
```

Appendix III

Templates for quantifiers.

```
template <typename In, typename UnaryOp>
bool quantifier(In first, In last, UnaryOp op, bool b)
{
  while (first!=last)
    if (!op(first++))
      return !b;
  return b;
}

template <typename Container, typename UnaryOp>
bool forall(const Container& c, UnaryOp op)
{
  return quantifier<typename Container::const_iterator, UnaryOp>(c.begin(),
c.end(), op, true);
}

template <typename Container, typename UnaryOp>
bool exists(const Container& c, UnaryOp op)
{
  return quantifier<typename Container::const_iterator, unary_negate<UnaryOp>
>(c.begin(), c.end(), unary_negate<UnaryOp>(op), false);
}

template <typename Container, typename UnaryOp>
bool exists1(const Container& c, UnaryOp op)
{
  typename Container::const_iterator first=c.begin();
  int count=0;
  while (first!=c.end())
    count+=op(*first++);
  return count==1;
}
```

Appendix IV
Sequence Comprehension

```
template <typename Generator, typename InputSet, typename ResultSeq, typename
Predicate>
struct SeqCompMemberGenerated : ternary_predicate<Generator, typename
ResultSeq::const_iterator, Predicate, InputSet> {
  SeqCompMemberGenerated(const Generator& generator, typename
ResultSeq::const_iterator r, const Predicate& predicate) :
ternary_predicate<Generator, typename ResultSeq::const_iterator, Predicate,
InputSet>(generator, r, predicate) {;}
  bool operator()(typename InputSet::const_iterator i) const { return
object2(*i) && object(*i)==*object1; }
};

template <typename Generator, typename InputSet, typename ResultSeq, typename
Predicate>
struct SeqCompExistsGenerator : ternary_predicate<Generator, InputSet,
Predicate, ResultSeq> {
  SeqCompExistsGenerator(const Generator& generator, const InputSet& inputSet,
const Predicate& predicate) : ternary_predicate<Generator, InputSet, Predicate,
ResultSeq>(generator, inputSet, predicate) {;}
  bool operator()(typename ResultSeq::const_iterator r) const { return
exists(object1, SeqCompMemberGenerated<Generator, InputSet, ResultSeq,
Predicate>(object, r, object2)); }
};

template <typename Generator, typename ResultSeq, typename InputSet>
struct SeqCompResultExists : binary_predicate<Generator, typename
InputSet::const_iterator, ResultSeq> {
  SeqCompResultExists(const Generator& generator, typename
InputSet::const_iterator i) : binary_predicate<Generator, typename
InputSet::const_iterator, ResultSeq>(generator, i) {;}
  bool operator()(typename ResultSeq::const_iterator r) const { return
object(*object1)==*r; }
};

template <typename Generator, typename ResultSeq, typename InputSet, typename
Predicate>
struct SeqCompProperty : ternary_predicate<Generator, ResultSeq, Predicate,
InputSet> {
  SeqCompProperty(const Generator& generator, const ResultSeq& resultSeq, const
Predicate& predicate) : ternary_predicate<Generator, ResultSeq, Predicate,
InputSet>(generator, resultSeq, predicate) {;}
  bool operator()(typename InputSet::const_iterator i) const { if (object2(*i))
return exists(object1, SeqCompResultExists<Generator, ResultSeq,
InputSet>(object, i)); else return true; }
};

template <typename Generator, typename InputSet, typename ResultSeq, typename
Predicate>
bool seqComp(Generator generator, InputSet inputSet, ResultSeq resultSeq,
Predicate predicate)
{
  return forall(resultSeq, SeqCompExistsGenerator<Generator, InputSet,
ResultSeq, Predicate>(generator, inputSet, predicate))
   && forall(inputSet, SeqCompProperty<Generator, ResultSeq, InputSet,
Predicate>(generator, resultSeq, predicate));
}
```

Appendix V
Encoding the mesh invariant without library support.

```
bool Mesh::inv() const
{
  for (vector<Node>::const_iterator p=meshNodes.begin(); p<meshNodes.end(); ++p)
{
    for (vector<Node>::const_iterator q=p+1; q<meshNodes.end(); ++q)
      if (*p==*q)
        return false;
    bool isAnElement=false;
    for (vector<Element>::const_iterator q=meshElements.begin();
q<meshElements.end(); ++q) {
      for (vector<NodeID>::const_iterator r=q->nodeIDs.begin(); r<q-
>nodeIDs.end(); ++r)
        if (q->deref(*r)==*p) {
          isAnElement=true;
          break;
        }
      if (isAnElement)
        break;
    }
    if (!isAnElement)
      return false;
  }
  for (vector<Element>::const_iterator p=meshElements.begin();
p<meshElements.end(); ++p) {
    for (vector<NodeID>::const_iterator q=p->nodeIDs.begin(); q<p-
>nodeIDs.end(); ++q) {
      if (find(meshNodes.begin(), meshNodes.end(), p-
>deref(*q))==meshNodes.end())
        return false;
      if (getAngle(p->deref(p->prev(*q)), p->deref(*q), p->deref(p-
>next(*q)))>=p->maxAngle())
        return false;
    }
    for (vector<Node>::const_iterator q=meshNodes.begin(); q<meshNodes.end();
++q)
//  it is an error if the node q is not in element *p and the node is enclosed
by the element
      if (!find(p->nodeIDs.begin(), p->nodeIDs.end(), q->id()) && p->encloses(q-
>id()))
          return false;
  }
  return true;
}
```

Appendix VI
Function object definitions by specialisation for the invariant of Mesh.

```
bool meshInv1::operator()(vector<NodeID>::const_iterator n) const
{ return object.member(*n); }

bool meshInv2::operator()(vector<Element>::const_iterator e) const
{ return forall(e->nodeIDs, meshInv1(object)); }

bool meshInv3::operator()(vector<Node>::const_iterator n) const
{return exists(object.meshElements, meshInv4(object)) && \
         forall(object.meshNodes, meshInv5(object, n)); }

bool meshInv4::operator()(vector<Element>::const_iterator e) const
{ return exists(e->nodeIDs, meshInv1(object)); }

bool meshInv5::operator()(vector<Node>::const_iterator e) const
{
  if (e==object1)
    return *e==*object1;
  else
    return *e!=*object1;
}
```

1 Bertrand Meyer. Object-Oriented Software Construction, 2nd Edition. Prentice Hall, 1997.

2 E. H. Dürr & J. van Katwijk. VDM++, a formal specification language for object-oriented designs. Proceedings of TOOLS Europe '92. Prentice Hall. 1992.

3 C++ International Standard ISO/IEC 14882, Section 14.7.3. ANSI 1998.

4 Bjarne Stroustrup. The C++ Programming Language, 3rd Edition. Addison-Wesley. 1998.

5 H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions, volume 551 of Lecture Notes in Computer Science, 428--456. Springer-Verlag, New York, N.Y., October 1991.

6 James McKim. Class Interface Design. Proceedings of TOOLS Pacific. Prentice Hall. 1995.

7 McKim J. Programming by contract - designing for correctness. JOOP, 9(2). 1996.

8 McKim and Mondou. Class Interface Design: Designing for Correctness. Journal of Systems and Software, 23 (2):85-92.

9 D. Hoffman. On Criteria for Module Interfaces'. IEEE Transactions on Software Engineering, pp 537-542, May 1990.

10 Mitchell R, Howse J and Hamie A. Contract-oriented specifications. in Chen J, Mingshu L, Mingins C and Meyer B (editors). Proceedings of TOOLS 24. IEEE. 1997

11 Leavens G. An Overview of Larch/C++: behavioral Specifications for C++ Modules. Technical Report TR#96-01c, Department of Computer Science, Iowa State University. 1997.

12 Gary T. Leavens & Yoonsik Cheon. Preliminary Design of Larch/C++. In U. Martin & J. M. Wing, editors, Proceedings of the First International Workshop on Larch, Dedham, July 1992. Springer Verlag.

13 Gary T. Leavens and Albert L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In Jeannette Wing and James Woodcock, editors. FM'99: World Congress on Formal Methods in Development of Computer Systems, Toulouse, France, September 1999. Lecture Notes in Computer Science, Springer-Verlag, 1999.

14 Jean-Marc Jézéquel, Michel Train, Christine Mingins. Design patterns with contracts. Addison-Wesley Reading, MA 1999.

15 Gary M. Weiss and Johannes P. Ros. Implementing Design Patterns with Object-Oriented Rules. JOOP November/December 1998.

16 John Barton, Lee Nackman. Scientific and Engineering C++. Addison Wesley. 1994.

17 James McKim. Class Interface Design. TOOLS Pacific 1995.

18 Cliff B. Jones. Systematic Software Development using VDM, 2nd Edition. Prentice Hall. 1980.

19 J. Michael Spivey. The Z Notation: A Reference Manual, 2nd edition. Prentice-Hall, Englewood Cliffs, N.J., 1992.

20 IFAD documentation. www.ifad.dk

21 David Maley, Ivor Spence. Emulating Design by Contract in C++. TOOLS-29 (Europe), editors Richard Mitchell, Alan Cameron Wills, Jan Bosch, Bertrand Meyer p66-75. IEEE. 1999.

22 David Maley, Ivor Spence. Supporting Design by Contract in C++ with Subtyping. Journal of Object-oriented Programming. 2000 (to appear).

23 S. Kent & Y. Maug. Quantified Assertions in Eiffel. Proceedings of TOOLS Pacific 95. Prentice Hall, Hemel Hempstead (UK). 1995.

24 Miguel Katrib & Jesús Coira. Improving Eiffel Assertions Using Quantified Iterators. JOOP Nov/Dec p35-43. 1997.

25 Bertrand Meyer. Applying Design by Contract. Computer (IEEE) vol 25 no. 10 p40-51 October 1992.