Config: a case study in combining software engineering techniques

David Maley[a] and Ivor Spence[b]
[a]*St. Mary's University College, 191 Falls Road, Belfast BT12 6FE, Northern Ireland, UK*
E-mail: d.maley@stmarys-belfast.ac.uk
[b]*Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland, UK*
E-mail: i.spence@qub.ac.uk

Config is a software component of the Graphical R-Matrix Atomic Collision Environment. Its development is documented as a case study combining several software engineering techniques: formal specification, generic programming, object-oriented programming, and design by contract. It is specified in VDM++; and implemented in C++, a language which is becoming more than a curiosity amongst the scientific programming community. C++ supports object orientation, a powerful architectural paradigm in designing the structure of software systems, and genericity, an orthogonal dimension to the inheritance hierarchies facilitated by object oriented languages. Support in C++ for design by contract can be added in library form. The combination of techniques make a substantial contribution to the overall software quality.

1. Introduction

The stages of the software development lifecycle are well-documented [1,2]. Improved techniques for the various stages of the process include the use of formal notation for the specification stage, object-orientation for the design and implementation stages, and design by contract for the testing stage. There is no implication intended that these stages follow in sequence – another overall improvement in software design methodology has been the development of a more integrated approach. In addition, the exploitation of genericity has proven to be a valuable technique at several stages of the software engineering process. This paper presents a case study in the use and interaction of these various techniques; the focus is on genericity, as a technique that has perhaps received less acclaim that it deserves in the literature to date.

Scientific Programming 8 (2000) 59–71
ISSN 1058-9244 / $8.00

2. Overview

This paper describes how the software engineering techniques of generic programming, object-oriented programming, formal specification and design by contract were employed in tandem in the construction of the software component Config [3], with a particular focus on the utility of genericity.

2.1. Genericity

Generic programming has the potential to enhance software quality in a number of ways. In its simplest form it is a way of avoiding duplicating code in circumstances where the only distinction between two pieces of code is the types they manipulate. Thus to use a list of INTEGERs and a list of REALs, only one list module (or whatever construct is appropriate) is needed. Software quality is in this case enhanced by elimination of duplication: this means that information is in one place only, and it therefore cannot be inconsistent. Then if it needs to be changed, there is no danger of missing an instance where the change is necessary. It also promotes reuse, since generic modules can be written which can serve a wide range of purposes when instantiated. This was the motivation for the C++ Standard Template Library (STL) [4]. Therefore, the designer and implementer can concentrate on the specific problems of the task at hand without having to reinvent the wheel each time a list of objects or a search algorithm is needed.

In this paper we demonstrate some more sophisticated uses of genericity, whereby two distinct aspects to supporting run-time assertion monitoring in C++ are illustrated: providing a framework in which the behavioural properties can be monitored, and providing a framework in which behavioural properties can be expressed.

Generic programming is a static technique, amenable to both object-oriented and non object-oriented programming. This is borne out in the languages that support it. For example, object-oriented C++ and non

object-oriented Ada do support it, and object-oriented Java and non object-oriented Fortran90 do not. It will be demonstrated in this paper how powerful and elegant techniques for expressing a software architecture are obtained from a language that supports both genericity and inheritance. Therefore genericity is used to best effect in an object-oriented environment. The implementation language chosen for the software component documented here, Config, was C++ [5]. A rich and diverse language, C++ now boasts both a standard and near enough standard-compliant compilers.

In practice, more often than not generic programming takes a form known as constrained genericity. It then becomes an abstraction technique, like the inheritance taxonomies of object-oriented languages. The abstraction in this case is a set of requirements on the data types that may be used to instantiate a generic parameter.

This case study takes examples from the development of Config, a software component of the Graphical R-Matrix Atomic Collision Environment [6]. Software for generating electron distributions and couplings was available in the computation stage of an early version of the R-matrix program package. However, the original package is written in FORTRAN 77 and is typical of many similar such large software systems, in that it has evolved over many years, through the efforts of numerous personnel, in a process of controlled iterative enhancement. Often the modification and extension of such large, complex and mature software systems is severely hampered because some components of the system are written in an implementation-dependent fashion, they are inadequately documented, their functionality is not precisely known, and under certain circumstances they fail to operate correctly.

The design goals of Config therefore included the following:

- write a formal specification of the component in order to state precisely and unambiguously the functionality of the component;
- apply object-orientation as the principal structuring mechanism for the system;
- make maximum use of existing libraries in order to minimise development time, and take advantage of the efforts and expertise of other developers;
- minimise the gap between the formal specification statement and the implementation. Ideally this means constructing a proof that the implementation is correct with respect to the formal specification; but if that is unrealistic then to adopt an intermediate methodology such as design by contract.

Initial work to address these problems for Config, involved developing a formal specification of the problem of generating electron distributions and couplings [7]. A version of the formal specification, rewritten to take advantage of the newly-emerging object-oriented specification language VDM++ [8] and showing the significant characteristics of the abstract data types of Config, is reproduced in WebAppendix I.

The formal specification states the properties of the data types involved, and the properties of the operations on them. It serves a number of important purposes. Firstly, it provides a precise, unambiguous, consistent and complete statement of the functionality of the component. At the same time, it does not over-specify: it states what the component should do, but without stating how it should do it. A finished program meets most of the criteria for being considered a formal specification: except that it over-specifies. A formal specification would state that a routine returns the square root of its input, but it need not state how the square root is to be computed. The original formal specification of Config was written in the language VDM-SL [9]. The second important purpose served by the formal specification is that it provides a basis for program proving, for reasoning about properties such as completeness and consistency, and for thinking in general. Thirdly it serves as system documentation, so maintainers are not obliged to read the code in order to discover the system's functionality. It has been shown that the predicates in a specification can be used as component-library search keys [10], and also that specifications can be used for test generation [11–14], animation [15], performance tuning and system integration.

VDM++ provides a number of types for modelling the problem domain, such as sets, maps and lists (lists are called sequences in VDM++). It is a natural next step to use the C++ Standard Template Library to implement these types. The types are known collectively in the STL as containers, since their purpose is to contain collections of objects. The STL provides generic container classes, from which new types can be constructed without having to be concerned with the implementation of the container data structures. Likewise, many of the common operations upon these containers are provided in the form of parameterised algorithms. These algorithms are in addition to the methods of the container classes. Indeed, some commentators would view the STL as a collection of generic algorithms, and consider containers as being provided simply to give the algorithms something to work upon.

The similarities between the data structures and structure manipulation methods provided by VDM++,

and those provided by C++ STL may lead one to wonder whether the VDM++ is really necessary. Can the STL instantiations not serve as specification, implementation and documentation? This is not the case, or at least not entirely. It is in the statement of the behaviour of the operations that constitute the model of the physical system where VDM++ provides abstractions that are not possible in C++/STL. In theory, C++/STL data structures and structure manipulation operations could be used as the basis for stating the operations that model the physical system (Larch [16] would permit this, for example), but it would not be possible to achieve the clear, precise and implementation-independent statement possible in VDM++. Furthermore, the abstract behavioural properties can then be transformed into validity checks as part of the design by contract process, independently of the algorithmic implementation of those properties.

One of the underlying principles of the STL is that complexity guarantees are given with the components. This means that operations are guaranteed to complete within stated time scales: it is recognised that a library will not be used if performance is sacrificed for the sake of generality. The recent C++ Standard [17] was even influenced by this requirement [18], with the addition to the templates section of partial specialisation, which allows algorithms and classes to be much more efficient. Partial specialisation adds a large degree of flexibility to working with the STL.

The STL also provides specialised support for numeric operations. For example, it is recognised that much numeric work relies on relatively straightforward single-dimensional arrays of floating-point values, and that these structures are supported by high-performance machine architectures and aggressive code optimisation. Consequently the STL provides a vector (called `valarray`) designed specifically for speed when using the usual numeric vector operations.

It has already been mentioned that the container class structures and their methods closely mirror the abstract data types used in the specification language. However, the STL does not provide mechanisms for checking the properties of the types created from it as they might be stated in a formal specification. A formal specification of the data types in a software component will specify not only data structure and permitted operations, but also constraints. In a language such as VDM++, for the data structures these will be in the form of invariants on the values of the data; for operations, they will be in the form of preconditions and postconditions on the execution of operations. Thus, the STL needs to be extended to accommodate this requirement.

It must be understood that the requirement to be able to monitor the constraints stated in the formal specification is not merely a luxury desired by computer science theorists: it is a basic tenet of the development technique known as Design by Contract [19], which has significant implications for software quality. Probably the foremost exponent of this technique is Bertrand Meyer, the designer of the language Eiffel [20]. Eiffel provides language support for Design by Contract. However, the language has had little impact thus far on the scientific community. One significant limitation of Eiffel is that the mechanism for expressing constraints is limited, and only simple propositions are permitted: certainly the assertion mechanism provided is not sufficient for many of the constraints of Config.

So what benefits does Design by Contract offer? In essence, the method is an aid to constructing correct programs. It also serves as a tool for documenting software, enhances the opportunity for reuse, and provides a debugging tool. It relies on the idea of assertions: predicates which express properties of the system. If a predicate is not satisfied, there has been an error. Monitoring assertions means that testing can be undertaken in parallel to implementation, not just as a subsequent phase. The source of an error is easiest to identify if the error is detected immediately following its introduction into the system; also, if development is suspended until the error is removed, the problem does not become buried under layers of additional complexity. Taking time to write a formal specification of the abstract data types in the system (that is, data structures and the operations upon them), coupled with the use of assertion checking during implementation, mean that overall development times and error rates can be reduced for the lifetime of the system [21].

Finally, it must be stated that program correctness is a relative term: a program is correct (or otherwise) only with respect to its specification. A correct implementation of an incorrectly-specified system will not exhibit the desired behaviour.

2.2. Basics of Config

The essential operation of the Config component is shown in Fig. 1. The component takes an atomic configuration, provided by the client, and a data table, which is fixed, and uses them to generate the electron couplings. The resulting data is termed a configuration tree list. A simple configuration tree is shown in Fig. 2.

Configuration trees have two significant characteristics, the symmetry of their root term ($^5G^o$ in the exam-

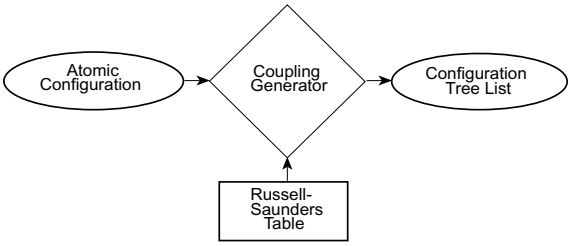

Fig. 1. Config functionality.

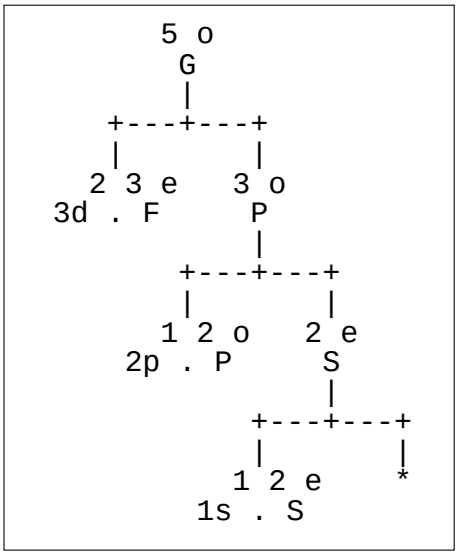

Fig. 2. A Configuration Tree.

ple) and their electron distribution ($1s^1 2p^1 3d^2$ in the example). Other components of the GRACE suite require a client to provide as input both the symmetry and the electron distribution of any trees being processed. However, it may or may not be possible to construct a tree exhibiting both a given symmetry and a given electron distribution. Config is used to prevent any inconsistencies of this nature arising: it can determine the symmetries of all the trees possible from a given atomic configuration and electron distribution, and all the electron distributions which give rise to a given symmetry from a given atomic configuration.

2.3. Object orientation and specification languages

Two of the most common languages used in formal specification are VDM-SL and Z [22], neither of which is object-oriented. For example, in VDM-SL, an uncoupling operation might be stated as below. Firstly, there is a specification statement which states that `ConfigurationTree` is defined to be a sequence of `ConfigurationTreeElements`.

ConfigurationTree = **seq of** *ConfigurationTreeElement*

Following this, the specification statements first state the signature of the operation, that it takes a `ConfigurationTree` as input and produces a `ConfigurationTree` as output, and then state the definition of the operation using one of VDM-SL's built-in operators for sequences: the VDM-SL operator **tl** removes the first element of a sequence. VDM-SL provides a number of ways to define the behaviour of an operation, and VDM++ provides still more. The form shown is known as definitional.

uncoupleConfigurationTree: ConfigurationTree → *ConfigurationTree*
uncoupleConfigurationTree(*ct*) $\triangleq$ **tl** *ct*

Object orientation is a software engineering technique whereby the construction of a system is based around varying types of objects. These objects provide functionality through a stated set of operations. If the object stores data it is often encapsulated in such a way that it can only be accessed via the operations. Thus the executing system may be viewed as a set of communicating objects, wherein objects send messages to one another requesting that stated operations be performed (perhaps with given data).

Each object defines precisely the format of the messages it is capable of responding to (its operations). Therefore, a list object might be capable of responding to a message that requests that a given item be appended to it, but is unlikely to be capable of responding to a message that requests the list to return its date of birth. This latter scenario is unlikely in a statically-typed language like C++, where the compiler can enforce correct protocol; but in other object-oriented languages, such as Smalltalk, any message can be sent to any object, leaving with the message sender the responsibility for ensuring that protocol is observed. The essential point is that each message is intended for and oriented around a particular type of object.

Both VDM-SL and Z now have object-oriented extensions (VDM++ and Z++ [23] respectively). VDM++ was chosen because the facility in VDM-SL for the explicit statement of preconditions and post-conditions integrated well with the desire to use design by contract as a basis for testing. The major differences between VDM-SL and Z are discussed more

fully in [24]. Larch/C++ [25] was also considered, since it offered the prospect of a rich specification library (the Larch Shared Library) and also greater expressivity of implementation concerns. However, the absence of tool support for typechecking made it an impracticable choice.

These extensions are constantly evolving, and the debate about how best to incorporate genericity into VDM++ is on-going; it is unfortunate that the current state of the language does not currently permit the economy of expression that can be enjoyed in C++. However, using VDM++, the object supplies a context that at least renders the name repetition in the above example redundant. Clearly there is a lot of repetition of the name `ConfigurationTree` in the definition of the uncouple operation, and using an object-oriented language eliminates this. This hardly constitutes a rationale for adopting an object-oriented approach, but the example typifies the way many operations are inherently oriented around a particular data item, in this case a `ConfigurationTree`. Hence in VDM++, a `ConfigurationTree` class which maintains a list `k` of `ConfigurationTreeElements` can be defined with an `uncouple`() member as follows:

```
class ConfigurationTree
instance variables
k:  seq of ConfigurationTreeElement;
...
operations
uncouple():  →
   pre len(k) >0
   post k = tl k∼
```

The signature of the operation is in this case empty, and the action is defined in terms of a postcondition which states the effect of the operation on the instance variable `k` of the `ConfigurationTree` object. The postfix tilde (∼) denotes the 'old' value of the sequence `k`, i.e. the value of the sequence prior to the operation.

2.4. The STL

The STL is a large and extensible body of efficient, generic and interoperable software components. Many of the fundamental algorithms and data structures of computer science are included, but with a crucial distinction from the way that they are usually found in library form: the algorithms and data structures are decoupled from one another. The STL can be thought of simply as a library of generic algorithms; the container classes of the library are there merely to give the algorithms something to work on. It is perfectly acceptable (and indeed expected) that STL algorithms will be used on user-defined containers, and that STL containers will be acted upon by user-defined algorithms. Adoption of constrained genericity permits the level of decoupling necessary to achieve this, using iterators and function objects. The STL algorithm `find_if(..)` illustrates their use.

```
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.

A key idea of the standard containers is that they should be logically interchangeable wherever reasonable, so that, for example, code written for a `vector<T>` may not need to be altered if a `list<T>` is used instead (however, the performance may alter). In order to achieve this, the STL makes use of some new programming concepts. Principal amongst them

is the idea of iterators: the purpose of an iterator is to provide a way to sequentially access the elements of an aggregate object without exposing its underlying representation. Moreover, they make it possible to traverse the container in different ways (such as from the end to the beginning). Iterators make it possible to decouple the algorithms and data structures. They are an example of a design pattern [26]. Design patterns are high level abstractions that can be adopted in the solution of various commonly encountered problems in object-oriented design.

Not every class can be used to instantiate `vector <T>`. For example, the instantiating class must provide a default constructor. When there is a requirement like this on the instantiating types, the genericity is known as constrained genericity.

`vector<T>` is used a number of times in Config, but in fact more often that not what is required is in fact an ordered vector. An ordered vector can be created from an unordered vector, and there is a discussion on how best to achieve this in WebAppendix II.

3. Example Config types

3.1. OrderedShellConfigurationList

In order to consider an `OrderedShellConfigurationList`, it is necessary to first consider the one fixed data item in Config, the Russell-Saunders table. A contraction of the Russell-Saunders table is shown in WebAppendix III. The properties of this data abstraction are stated in the formal specification. It is reasonably straightforward to map the data structures stated in the specification to instantiations of data structures provided by the STL. However, there is no such direct correlation with the stated behavioural properties for the physical operations.

The Russell-Saunders table is used to create an `OrderedShellConfigurationList`, which is an ordered list of `ShellConfigurations`. All the members of an `OrderedShellConfigurationList` have identical `ShellDescription` components, and are ordered by their `ShellTerms`. The `ShellTerms` are obtained from the table. Thus an example of an `OrderedShellConfigurationList` would be

$$2p^4\ {}^1_0S^e \quad 2p^4\ {}^3_2P^e \quad 2p^4\ {}^1_2D^e$$

In Fortran90 an implementation of an `OrderedShellConfigurationList` would need to provide operations such as `create_empty, length, head, tail, copy` and `append` (the full implementation is given in WebAppendix IV). The choice of a linked list implementation (requiring the additional functions `mallocate` and `dispose`) is not essential for an `OrderedShellConfigurationList` since there is a known upper bound on its length, but the choice is not crucial to the illustration. There are two points to note about this implementation.

(i) the major point is that the implementation of a number of other modules in the component, such as `ShellList`, `ConfigurationTree` and `ConfigurationTreeList` (not to mention list modules in other components) would look practically identical. They must also provide operations such as `create_empty, length, head, tail, copy` and `append`. This repetition is tedious and unnecessary – if not error prone – for the programmer. In addition, it is not readily amenable to change, and the larger the scale of the change, the worse the situation becomes. If a bug is found in a list routine, an amendment is required for each of the list types used. If it is found that the list implementation is inefficient and must be changed in its entirety, the amount of work increases in proportion.

(ii) a less significant point is that the programmer has to be concerned with handling dynamic storage. A different choice of representation which does not use dynamic storage can be made for `OrderedShellConfigurationList`, but not for the other list types in Config, so the issue cannot be avoided. Handling dynamic storage can be a serious issue which can impair implementation performance, not to mention programmer productivity. One immediate consequence is that if routines are defined to return user-defined types, then unavoidable memory leakage can result [27].

Using the STL, both of these problems are immediately overcome. To define an `OrderedShellConfigurationList`, once the list element type `ShellConfiguration` has been defined, an `OrderedShellConfigurationList` can be defined by instantiating the container class `vector<T>` with the type `ShellConfiguration`:

```
class OrderedShellConfigurationList:
   public vector<ShellConfiguration>
   {
```

```
public:
   OrderedShellConfigurationList
   (ShellDescription);
};
```

The STL container class `vector<T>` provides practically all the functionality required to implement an `OrderedShellConfigurationList`, such as `insert(..)`, `erase(..)` and `size()` operations. Only the construction functionality is missing. In C++, when an object is created its constructor is called. Constructors can be overloaded, and so the parameters given at the point of creation determine which constructor is called. If no constructors are given, the compiler will generate empty constructors. In the case shown, an `OrderedShellConfigurationList` is constructed from a `ShellDescription` (and the `RSTable`, which need not be a parameter to the constructor since it is a constant). In order to provide this additional constructor the class must inherit from `vector<ShellConfiguration>` (and not just `typedef` it).

However, there is another aspect of the specification of `OrderedShellConfigurationList` besides construction that the STL does not provide. The specification also states an invariant. This is a property that any `OrderedShellConfigurationList` must observe in order to be valid.

The invariant states that the `ShellTerms` of the elements of an `OrderedShellConfigurationList` mirror those in the RS-Table, and that the `ShellDescriptions` in a given row are all identical. It also makes a statement about the symmetry of the `RSTable`, and states a simplifying assumption about shells with `OrbitalAngularMomenta` that are greater than or equal to three. This last point reflects the remark made at the end of the introduction, that correctness is a relative notion.

This invariant must hold both before and after every operation on an `OrderedShellConfigurationList` object. It must also hold after construction: it is therefore an implicit postcondition on the constructor. However, it does not state how the construction process is to be carried out: it is simply a predicate on the state of the object when construction has been completed. In a development environment, ideally one implementor writes the constructor and another writes an implementation of the invariant that checks that the constructor is creating valid objects.

Notice that the code that has to be written – the constructor – is precisely that which is particular to the system. The writer of the STL can provide operations such as appending, and erasing, but is unlikely to know about, and certainly cannot cater for, the construction of an `OrderedShellConfigurationList` from a `ShellDescription` and the `RSTable`: but that is not necessary. The power of inheritance and genericity combined mean that very often there is an appropriate and unique point for each piece of programming that is specific to the component. Not only can the implementer concentrate on the specific problems of the task in hand without having to reinvent the wheel each time a list of objects is needed, better still very often there is no need to invent the wheel in the first place since it has already been invented by someone else.

The implementation of an `OrderedShellConfigurationList` therefore amounts to writing the constructor, which must ensure that the constructed object satisfies the invariant. Therefore it is necessary to have the formal specification available in order to inform the writing of the constructor (and also to inform the independent writing of the code that verifies the properties of the constructor).

3.2. ConfigurationTreeList

A `ConfigurationTreeList` is declared as follows:

```
class ConfigurationTreeList:
public vector<ConfigurationTree>
{
   public:
   ConfigurationTreeList
   (ConfigurationTree,
   OrderedConfigurationList);
   ConfigurationTreeList
   (AtomicConfiguration);
};
```

It therefore follows the same pattern as `OrderedShellConfigurationList`, of inheriting a container instantiation and adding functionality. In this case the functionality is in the form of two constructors.

Most of the operations on a `ConfigurationTreeList` are indistinguishable from those on other lists in the system (and any system), and these are implemented simply by picking them off the shelf by instantiating a generic component. The code that is specific to a `ConfigurationTreeList` has one and only one place in the system, within the two constructors, which encapsulate all the work involved in generating a `ConfigurationTreeList` from an `AtomicConfiguration`.

3.2.1. STL algorithms

It is important to bear in mind that whatever form of ordered container is used for a `ConfigurationTreeList`, the term 'ordered' simply refers to the fact that the ordering is a property that is expected to be displayed by the container: it is not maintained by it. (It is of course possible to create such a container, whereby each operation ensures that ordering is maintained). However, in Config, it is the responsibility of the container instantiator to ensure that ordering is maintained. In the case of the construction of a `ConfigurationTreeList`, the `ConfigurationTrees` are not appended to the list in the desired order. Consequently it is necessary to sort the list to put the trees in order after generation. This step highlights the power of the STL. To sort the `ConfigurationTreeList ctl`, it is only necessary to write:

```
sort(ctl.begin(), ctl.end());
```

Clearly for this to be possible, constrained genericity is again at work. The sort algorithm in the STL relies on an ordering operator $<$ being defined for `ConfigurationTree`, or unambiguously for some type `T` on the inheritance chain of `ConfigurationTree`. Apart from this, the sort algorithm knows nothing else about `ConfigurationTrees`, and all it knows about the container they are in is that it observes the iterator protocol. If a `ConfigurationTreeList` was reimplemented using a `list<T>` instead of a `vector<T>`, the sort line in the code would not need to be changed. It is said to be 'loosely coupled'.

4. Supporting design by contract

4.1. The notion of contract

Human contracts involving two parties are characterised by two major properties:

- each party is persuaded to enter into the contract by the benefits on offer; they accept that along with privilege comes responsibility, and are willing to therefore incur some obligations to obtain the benefits;
- these benefits and obligations are documented in a contract document. The contract document protects both parties. It protects the client by specifying the minimum that should be done: the client is entitled to receive a certain result. It protects the supplier by specifying how little is acceptable: the contractor must not be liable for failing to carry out tasks outside of the specified scope.

Clearly, what is an obligation for one party involved is usually a benefit for the other. Design by Contract applies the notion of contract to software design. Each routine states the contract it is willing to enter into with a client. It states a precondition, which must be satisfied by the client in order for the client to enjoy the benefits of the contract; and it states a postcondition, which details the benefits of the contract for the client. If either the precondition or postcondition of a routine are not met at any stage in the execution of the system, the contract has been broken and the error can be highlighted. Properties that must be maintained by all routines of a class can be grouped, and has become known as the class invariant.

One important aspect of Design by Contract is what happens to the contract when one class inherits from another. This issue (often cited as the Liskov Substitutability Principle [28]) is discussed by Meyer, and its emulation in C++ is dealt with in another paper by the current authors [29].

4.2. Supporting design by contract in general

The VDM++ language permits the precise specification of the properties of the Config 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. This section gives an example of using the generic libraries we have developed which help verify that the outcome of the honest toil is the same as that obtained by theft.

In order to achieve this in C++, it is necessary to provide a framework in which the behavioural properties stated in VDM++ can be monitored, and also provide a framework in which behavioural properties can be expressed. The current authors have demonstrated how a framework for the expression of preconditions and postconditions, along with class invariants, can be added to C++ classes by the instantiation of templates, and in such a way that the code of the class methods

remains unchanged [29]. We are now working on a framework for expressing the behavioural properties, and what is below is very much a snapshot of the current state of our work. The next section on adding predicates to the STL may be more accessible for those not familiar with generic programming.

Consider, for example, the VDM++ statement of the `ShellList::erase(index)` routine, which is stated like this.

```
class ShellList
instance variables
...
shells:  seq of ShellOccupancyRange;
...
operations
...
erase(index :  nat1)
ext rd shells
pre index<=len(shells);
post forall i in set inds(shells) &
i < index => shells(i)=shells~(i) and
i >= index => shells(i)=shells~(i+1)
and
len(shells)=len(shells~)-1;
...
end ShellList
```

The instantiation

```
typedef DBC0<ShellList, int> Erase;
```

within the scope of the `ShellList` class in C++ brings the function

```
bool ShellList::Erase::post(int)
```

into scope. The definitions of the templates ensure that this function is automatically evaluated after the `erase(..)` method has executed, and an exception is thrown if the stated condition is not met. This function has a generic definition, but this can be overridden by specialisation so that it mirrors the VDM++ statement, and this is done in this case as follows:

```
bool ShellList::Erase::post(int
index)
{
return forall(shells,
ShellListErasePost(*_object, index,
*_old))
}
```

The parameter `_object` is a pointer to the current object (i.e. in this case, a `ShellList`) which is brought into the scope of the `pre(..)` and `post(..)` routines by the instantiation of the generic class `DBC0<.>`. The parameter `_old` is a pointer to a (deep) copy of the object state prior the operation, again made automatically accessible by the instantiation. The definition makes use of the generic function `forall(..)`. We are developing a library of functions which facilitate the rapid transformation of VDM++ statements into executable predicates in C++. Quantifiers, comprehensions and sequence/set transformers are all amenable to this approach.

The `forall(..)` function takes a function object `ShellListErasePost(..)` as a parameter. This function object is a specialisation of the generic class `ternary_predicate<Object, T1, T2>`. 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 in this by providing them with a template class displaying some or all of the required properties, which can be specialised or inherited. The class `ternary_predicate<Object, T1, T2>` is therefore provided, (which itself inherits from the template class `unary_function<T, bool>`, which is provided by the STL).

The function object is defined

```
typedef ternary_predicate<ShellList,
int, ShellList>ShellListErasePost;
bool ShellListErasePost(int i) {
   int index(object1);
   ShellList old(object2);
   return ((i<index)?(object[i]==
    old[i]):
   (object[i]==old[i+1])) && \
   onject.size()==old.size()-1
}
```

One of the unfortunate consequences of using generic functions is that semantically significant information can be lost through the parameter-passing mechanism. We are still investigating the best approach to tackling this difficulty; in this case it has been reintroduced locally.

Note that iterators have not been used in this example. This is because the VDM++ refers to values in the current `ShellList` and the `ShellList` prior to the erase operation using the same index; but the same iterator would not be valid. This is another aspect of the technique that we are hoping to improve upon.

The use of function objects and generic routines involving iterators is an unfamiliar style of programming for many. However, it is a powerful abstraction technique and once mastered permits concentration on the specific details of the problem in hand and offers the potential for a great deal of code reuse .

4.3. Supporting design by contract using the STL – monitoring and managing objects

It has been shown how the STL provides the data structures, and many of the operations, needed to implement the specification of a component. However, the specification also states constraints on the data and operations, and the STL as it stands does not provide any way to incorporate these into the implementation. In order to ensure software quality, it is important that the constraints stated in the specification are monitored during the development of the software to ensure that the program behaviour is conforming to that stated by the specification writer.

In order to accommodate Design by Contract into the STL, it was necessary to find a way to extend the capabilities of the STL, but without interfering with the code. This can be done using the general approach outlined in the previous section, but there is an alternative approach which requires less familiarity with generic programming. This work is documented fully elsewhere [30], and only a brief description is given here. It is not necessary to be fully cognisant of how the mechanism works in order to take advantage of the enhanced STL.

In essence, the capabilities of the STL can be extended by using namespaces. A new version of the STL is written within a new namespace. The client code, occupying a third namespace, can remain unaltered, but its behaviour changes depending upon which namespace it imports the named features of the STL from. Thus, in a production version of the code, the client can choose to revert to the original STL to abandon constraint checking in favour of maximising execution speed.

The claim that the client code can remain unaltered needs some clarification. Clearly, in order to monitor constraints, the client must supply definitions of boolean functions that check the various assertions. However, it is not necessary to declare these in the class declarations: they are declared by the enhanced STL, and given default definitions. The client simply defines a specialisation of the required function, and the compiler adopts it in preference to the default. Thus, when reverting back to the original STL, these definitions must be removed since the functions that they specialise are no longer declared. This can be achieved by conditional compilation – there is no need for actual editing.

The container types of the STL are enhanced in a namespace `stdpp` by deriving new classes from the existing containers. These new classes can appear to have the same names as the original ones (of course, in fact, the names are different: for example, `::std::vector<T>` and `::stdpp::vector<T>`). Loose coupling ensures that the algorithms work on the new containers. `stdpp` can be used as an alternative to the STL by any client.

The methods of the class can also be redefined in such a way that they call the original methods, but check that the precondition is satisfied beforehand and the postcondition is satisfied afterwards. The client has the option of supplying specific definitions of preconditions and postconditions in the form of partial specialisations. Clearly granularity of control over which checks are employed can be introduced.

There are really two separate cases to consider. Firstly, there is the case of preconditions and postconditions which are inherent to a container operation because of the properties of the container. For example, it is an error to try to erase an element from an empty vector, so there should be a precondition on erasing from a vector stating that the iterator parameter lies in the range of iterators of the vector. Secondly, there are the precondition and postconditions which are specific to the instantiation of an operation for a particular type. Both can be accommodated.

Once constraint monitoring is employed, the question is immediately raised of what action should be taken when a constraint is found to be violated. This issue is beyond the scope of this paper. The default choice is to throw an exception.

4.3.1. Creating and using managers

In Config, the enhanced STL containers inherit the original container and a generic class. This generic class, `CoManager<T>`, provides the additional functionality required to create a type hierarchy which parallels the existing type structure of an object. This is a more sophisticated (and resource-hungry) approach than that described in the previous section. The type sub-structure of any object can be paralleled by a Manager structure which is automatically generated simply by instantiation of a Manager object. The sub-Managers are generated recursively by calls in the Man-

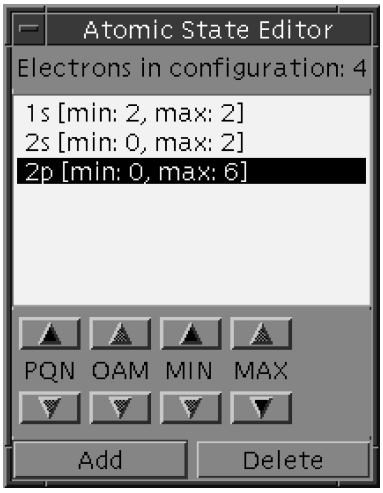

Fig. 3. Atomic state editor.

ager constructors; constructors are written for each type of container. These Managers creates a platform from which a number of services can be administered. Monitoring constraints is just one of these services; others are described below.

4.3.1.1. Editing structured types. Whether the user is permitted to enter values directly when editing a structured type, or whether the user must choose from a selection of pre-validated options, it is unlikely that an instance of the type in its entirety will be changed. It will be a component of the type that is under scrutiny, such as using the Atomic State Editor to increment a `PrincipalQuantumNumber` (Fig. 3).

The scheme is that where appropriate Managers provide an operation – `constructValue`() – that changes the value they manage. This change is propagated throughout the Manager hierarchy by a call to `constructValues(..)`. This calls `constructValue`() for the local value, and passes the call on to its parent. Disassembly and reassembly are modularised. The interface writer does not have to consider how to provide an operation to update the `PrincipalQuantumNumber` of a given `Shell` of a given `ShellOccupancyRange` of a given `ShellList` of a given `AtomicConfiguration` (and then repeat the process for `OrbitalAngularMomentum`). The Manager writer encodes the assembly of an instance of the Managed type from the existing value and one amended component into `constructValue`(). This process of assembly propagates up the structure through the hierarchy of Managers.

The process of disassembly is undertaken by supplying the structure navigation information. The process of reassembly is undertaken by a single call to `constructValues(..)`, which recurses up the Manager hierarchy.

4.3.1.2. Edit operation sensitivity. Considering again Fig. 3, it can be seen that some operations on the current `AtomicConfiguration` are sensitive – available to the user – and some are not. The button sensitivity reflects the constraints given in the specification: the user is not permitted to create an invalid instance of an `AtomicConfiguration`.

The sensitivity of a given button in the editor at a given time (i.e. in a given state) can be determined by fully stating the precondition of the operation. The precondition states for which values of the domain of the operation the postcondition is guaranteed. An implicit part of any precondition or postcondition is that the class invariant holds after the operation. The complete class invariant of a container object is made up of the conjunction of the invariants of its elements and the invariant specific to the container. When the action of the operation, and the desired postcondition are known, mathematically determining the precondition is a daunting task even in such a straightforward case.

A simpler approach is for the system to try the operation beforehand and see if an invalid object results. Of course, this is not an approach that can be adopted in all circumstances: 'erase_hard_disc', 'fire_missiles' are examples of operations where it would be undesirable to determine availability by this means. Clearly, it only works for readily-reversible operations, or when it can be performed on a harmless copy of the object in question. This is possible in Config, where the Manager structure readily facilitates this.

4.3.1.3. Data display prototyping. One of the aims of the GRACE project, which has been ongoing for a number of years, has been to provide a graphical user interface on UNIX workstations to computational physics software running on supercomputers. MOTIF [31] was chosen from the outset for this purpose, and subsequently it has become necessary to develop ways to use it with C++. Encapsulation of graphics elements into reusable classes is dealt with by Young [32] amongst others. The Manager and CoManager classes provide a platform for developing data display prototypes which interact well with MOTIF.

The MOTIF specifics needed to display each type of container class can be provided by classes such as

`vectorDisplay<T>` and `pairDisplay<pair<T1, T2>>`. These classes can be used to display all types used within Config, although particular requirements at times require special adaptations to be written, such as that used to display large `ConfigurationTrees`.

5. Conclusions

The traditional approach to component design has very often been functional decomposition. However, this model is not always congruent with the system under consideration, and conceptual integrity is lost. The object-oriented approach proposes object-based decomposition, and argues in addition that this approach will yield more stable units of reuse. One mechanism for facilitating this structuring process is inheritance, and is an essential characteristic of object-orientation. A second mechanism, which is largely orthogonal to the first, and is not restricted to object oriented languages, is genericity. An implementation language which provides both facilities offers the software engineer a very powerful structuring tool.

In order to take best advantage of these mechanisms, it is important to produce a precise statement of the nature of the component under consideration. Often, it takes bitter experience of the classic maxim, 'implement in haste, debug at leisure' to convince software writers of the benefits of investing in the early parts of the software design lifecycle. Basing design and implementation on formal specification is an approach which is invaluable when striving to improve software quality. A criticism sometimes made of formal specification is that specifications tend to become out-of-date and inconsistent with the implementation. However, we would argue that if the specification language is describing the data types which reflect a physical reality then these do not go out of date, and it is the implementation alone that runs the risk of inconsistency. The use of design by contract can help to highlight whenever such an inconsistency is introduced.

This paper focuses on the benefits of using an implementation language that supports generic expression. In the implementation phase, it is demonstrated that genericity enables the reuse of existing components. The STL provides both containers and algorithms that permit rapid implementation of precisely specified data types. Common functionality can be obtained 'off the shelf', enabling the implementor to concentrate on the aspects that are specific to the particular problem domain.

In addition to facilitating the reuse of existing components, genericity makes it possible for the implementor to capture commonalities within the current component, and to create reusable components. The intention may be for reuse within the same component or it may be wider afield.

The combination of VDM++ and C++ also permits predicates specifying component behaviour to be stated in an abstract mathematical language, and then encoded into checks that can be carried out at runtime. The likelihood of introducing errors during this encoding process can be minimised by constructing a library of functions in C++ that mirror the language constructs of VDM++. 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 powerful combination of inheritance and genericity provided by C++ make it possible to add a substantial degree of support for design by contract. In the absence of formal proof, this technique minimises the gap between the formal specification statement and the implementation. For whilst the data structures and the basic operations that manipulate them are similar in a specification language such as VDM++ and in a generic library such as the C++ STL, the use of a formal specification language permits the expression of properties that cannot be obtained 'off the shelf'. These must be encoded algorithmically by the scientific programmer; effort can be concentrated here because the basic data structures have been provided. The formal specification informs the design, the design directs the implementation, and genericity makes it possible to add support for checking that the implementation does indeed display the stated properties, adding to confidence in the quality of the product.

WebAppendices

http://www.stmarys-belfast.ac.uk/~ d.maley/
research/. . .
I. config1.rtf; II. OrderedVector.doc;
III. RSTable.doc; IV. F90Verbosity.doc.

References

[1] I. Sommerville, *Software Engineering,* (2nd ed.), Addison-Wesley, Wokingham, 1985.

[2] M. Jackson, *System Development*, Prentice-Hall, Eaglewood Cliffs, 1983.
[3] D. Maley, I. Spence and P. Kilpatrick, Config: A GRACE tool for constructing configuration trees. Computer Physics Communications Special Issue on continuum states of atoms and molecules, *Computer Physics Communications* **114** (1998), 271–294, Elsevier, North Holland.
[4] Alexaner Stepanov (Silicon Graphics Inc.) and Meng Lee (Hewlett Packard Laboratories), The Standard Template Library, 1995.
[5] B. Stroustrup, *The C++ Programming Language*, (3rd ed.), Addison Wesley, 1997.
[6] V.M. Burke, P.G. Burke and N.S. Scott, GRACE, *Computer Physics Communications* **69** (1992), Elsevier, North Holland.
[7] Scott, Kilpatrick and Maley, The Formal Specification of Abstract Data Types and their Implementation in Fortran 90, *Computer Physics Communications* **84** (1994), 201–225, Elsevier, North Holland.
[8] IFAD documentation, http://www.ifad.dk.
[9] Cliff Jones, *Systematic Software Development using VDM*, (2nd ed.), Prentice Hall, 1980.
[10] Eugene J. Rollins and Jeannette M. Wing, Specifications as search keys for software libraries, *Proceedings of the International Conference on Logic Programming*, 1991.
[11] Christophe Meudec, Automatic Generation of Software Test Cases from Formal Specification, *PhD thesis*, Queen's University of Belfast, 1998.
[12] M. Holcombe, An Integrated methodology for the specification, verification and testing of systems, *Journal of Software Testing, Verification and Reliability* **3**(3–4) (1993), 149–163.
[13] I. Hayes, Specification Directed Module Testing, *IEEE Transactions on Software Engineering* **12**(1) 124–133.
[14] L. Zucconi and K. Reed, Building Testable Software, *Software Engineering Notes*, Sep. 1996, pp. 51–55.
[15] J. Bicarregui, J. Dick, B. Matthews and E. Woods, Making the most of formal specification through animation, testing and proof, *Science of Computer Programming* **29** (1997), 53–78.
[16] J. Guttag and J.J. Horning, *Larch: languages and tools for formal specification*, Springer Verlag, 1993.
[17] C++ International Standard ISO/IEC 14882, Section 14.7.3, ANSI 1998.
[18] Alexander Stepanov, Interviewed by Al Stephens in Dr. Dobb's Journal, 1995.
[19] Bertrand Meyer, Applying Design by Contract, *Computer* (*IEEE*) **25**(10) (Oct. 1992), 40–51.
[20] Betrand Meyer, *Object-Oriented Software Construction*, (2nd ed.), Prentice Hall, 1997.
[21] Bennet P. Lientz and E. Burton Swanson, Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organisations, Addison Wesley, Reading (Mass.), 1980.
[22] J. Michael Spivey, *The Z Notation: A Reference Manual*, (2nd ed.), Prentice-Hall, Englewood Cliffs, NJ, 1992.
[23] Kevin Lano, *Formal Object Oriented Development*, Springer Verlag, 1995.
[24] I.J. Hayes, C.B. Jones and J.E. Nicholls, *Understanding the differences between VDM and Z: Technical Report UMCS-93-8-1*, Department of Computer Science, Manchester University, 1993.
[25] Gary T. Leavens and Yoonsik Cheon, Preliminary Design of Larch/C++, in: *Proceedings of the First International Workshop on Larch*, U. Martin and J.M. Wing, eds., Dedham, Springer Verlag, July 1992.
[26] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns for Object Oriented Software*, Addison Wesley, 1994.
[27] D. Maley, P.L. Kilpatrick, N.S. Scott, E.W. Schreiner and G.H.F. Diercksen, The Formal Specification of Abstract Data Types and their Implementation in Fortran 90: Implementation Issues Concerning the Use of Pointers, *Computer Physics Communications* **84** (1994), 201–225, Elsevier.
[28] Barbara H. Liskov and Jeannette M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems* **16**(6) (Nov. 1994, 1811–1841.
[29] David Maley and Ivor Spence, Supporting Design by Contract in C++ with Subtyping, *Journal of Object-oriented Programming* (2000), (to appear).
[30] David Maley and Ivor Spence, *Emulating Design by Contract in C++*, TOOLS-29 (Europe), Richard Mitchell, Alan Cameron Wills, Jan Bosch and Bertrand Meyer, eds., pp. 66–75, IEEE, 1999.
[31] Marshall Brain, *MOTIF – The Essentials and more*, Digital Press, 1992.
[32] Douglas Young, *Object-Oriented Programming with C++ and OSF/MOTIF*, Prentice Hall, 1995.