Emulating Design by Contract in C++

David 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

To date, much of the software written in Computational Physics has been produced with little regard paid to modern software engineering techniques; this paper documents experiences in beginning to address this shortcoming for Config, a component of the Graphical R-Matrix Atomic Collision Environment (GRACE).

The work is based around a formal specification of the Config component. The principal focus of the paper will be on a non-intrusive mechanism for monitoring constraints such as class invariants, preconditions and postconditions for highly structured data types based on the containers and algorithms of the Standard Template Library (STL), a mechanism which can be extended to handle structured object update and display prototyping.

Keywords: Computational Physics, C++, Standard Template Library, Design by Contract.

1. Introduction

Modern software development techniques such as formal specification and object orientation have had limited success in terms of take up by the scientific programming community. This paper outlines positive experiences in attempting to apply such techniques to a problem in Computational Physics.

The Config program [1] generates electron couplings from a given atomic state. The original package is written in Fortran 77 as part of the *R*-matrix program package [2]. 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.

Three of the design goals of Config were therefore the following:

- to write a formal specification of the component in order to state precisely and unambiguously the functionality of the component.
- to make maximum use of existing code in order to minimise development time, and take advantage of the efforts and expertise of other developers.
- to minimise the gap between the formal specification statement and the implementation. Ideally this would mean constructing a proof that the implementation was correct with respect to the formal specification, but if that proved to be unrealistic then to adopt an intermediate methodology such as Design by Contract.

Initial work to address these problems for Config therefore involved developing a formal specification, in VDM-SL [3], of the problem of generating electron distributions and couplings [4].

The formal specification 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. Work is ongoing to rewrite the specification using an object-oriented language, VDM++ [5]. (It is unfortunate that there appears to be widely varying degrees of support for the object oriented development at the various stages of the software life cycle. Whilst

there are sophisticated languages, plenty of texts on analysis and design [6, 7], and increasing amounts of work on testing, formal specification seems to be less in the limelight.)

The language chosen for the implementation was C++. Given the importance placed upon the formal approach, Eiffel [8] might appear to be a more attractive candidate. However, some of the constraints stated in the specification of Config are not immediately expressible in Eiffel's Design by Contract [9] support. The language chosen, VDM-SL, can be used to state any expression of the first order predicate calculus (as well as certain expressions of higher order calculi, such as whether a graph is cyclic). Although mechanisms such as universal quantification can be indirectly expressed in Eiffel, it is desirable that the construction of the formal specification should not be impeded by having to consider what device to employ in expressing a particular constraint. Another side of this argument is, of course, that not all VDM-SL specifications are implementable: but the coupling generation of Config was known to be computable since the Fortran 77 version already existed. In addition, it is shown that adding Design by Contract support for the STL [10] provides a platform for the provision of other services which are necessary to promote a faithful implementation of the specification.

The STL provides generic container classes and parameterised algorithms that closely mirror the abstract data types used in a specification language; however, the STL does not provide mechanisms for checking the properties of the types instantiated from it as they might be stated in a formal specification. In a language such as VDM-SL, for the data structures these are in the form of invariants on the values of the data; for operations, they are in the form of preconditions and postconditions on the application of operations. To achieve the design goals, it was therefore necessary to create some form of Design by Contract support in C++. Two separate (though complementary) techniques have been developed, and this paper documents the first, simpler, technique, which sufficed for the implementation of Config.

This paper describes a mechanism which automatically generates (simply by instantiation) an object hierarchy (the objects are known as *Managers*) which parallels the data sub-structure of an existing object, and which can provide a platform from which to administer tasks such as constraint monitoring, object update and object display.

In the first section of the paper, the essential functionality of Config is outlined. In order to illustrate the basic approach, an example from a more familiar realm is put forward, and it is shown how constraint monitoring can be achieved by a simple non-intrusive scheme. A more versatile scheme is outlined in the following section, based on a small class hierarchy of *Manager* classes. The Managers necessary to observe and report on the state of a highly-structured object can be automatically generated from the type structure of the object. The following section shows how the Manager scheme can be extended to provide a structured approach to editing the object and to data display prototyping. Finally, the limitations of the scheme are discussed. One significant limitation is that the mechanism outlined does not automatically observe the Liskov Substitutability Principle [11]. This has now been overcome by a second technique [12], and will be the subject of a separate paper.

2. The motivation for Config

The essential operation of the Config component is shown in Figure 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 Figure 2.

Configuration trees have two significant characteristics, the symmetry of their root term ($^5G^o$ in the example) and their electron distribution ($1s^1 2p^1 3d^2$ in the example). Other components of the G*R*ACE suite require a client to provide as input both the symmetry and the electron distribution of any trees being processed. However, it 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.

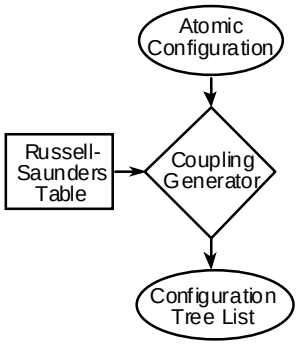

Figure 1. Config functionality.

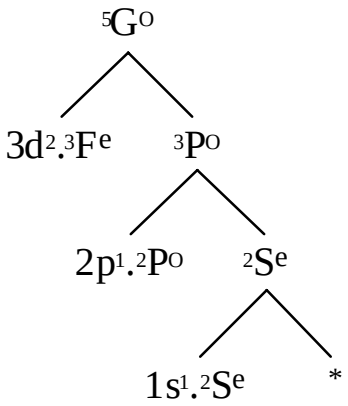

Figure 2. A Configuration Tree.

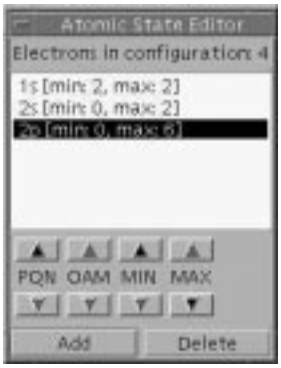

Figure 3. AtomicState Editor.

The vast majority of data types in Config are derived from the STL containers vector, pair and map. One of the main concerns in implementing Config was to ensure that the data supplied by the client – primarily an atomic configuration – conformed to the specification. An AtomicConfiguration is a list of ShellOccupancyRanges, together with a number of electrons. A ShellOccupancyRange is made up of a PrincipalQuantumNumber, an OrbitalAngularMomentum, a minimum Occupancy and a maximum Occupancy. The AtomicConfiguration editor is shown in Figure 3. It was a design goal of Config that the user should not be able to use this editor to create an invalid AtomicConfiguration: one which violates the constraints given in the specification. This paper documents how this was achieved, and in a way which can be readily applied to any instance of highly structured data.

3. Previous work on supporting Design by Contract in C++

The most straightforward approach to supporting Design by Contract in C++ is to use Eiffel for those components for which monitoring constraints is most important. However, as was mentioned in the introduction, the expressivity of the Eiffel assertion mechanism isn't always immediately sufficient.

There have been a number of different approaches to supporting Design by Contract in C++: use the existing language constructs to provide a support mechanism (the Percolation pattern [13]); use macros to support the inclusion and monitoring of constraints ([14], Nana [15]); effectively extend the syntax of the language by writing a pre-processor which converts non-C++ constraint expressions into C++ before compilation (App [16]); and language extensions (exlC++, [17], A++ [18]). Our approaches have used templates, utilising new features such as partial specialisation and inner template classes.

4. Constraint monitoring

Consider a simple example, in which the results of a player's matches in a tennis tournament are being entered into an appropriate data structure. In this example the results may be input from a keyboard or a data file; more often in a non-trivial situation, the data to be monitored will be the result of an operation. A player's results will be a series of Matches, each Match consisting of a series of Sets, and each Set consisting of two scores, the number of games won by each player in the Set. Clearly, there are potentially many instances of the class Results which would not in fact be a valid series of tennis scores. Using a formal specification language, the valid instances can be stated explicitly.

For example, an Eiffel-like version of TennisSet is given below.

```
class TennisSet creation
 make
feature -- Initialisation
```

```
 make(s1: INTEGER, s2: INTEGER) is
 do
  score1 := s1
  score2 := s2
 ensure
  score1_set: score1 = s1
  score1_set: score2 = s2
 end
feature -- Access
 score1 : INTEGER
 score2 : INTEGER
invariant
 at_least_six_to_win: score1>5 or score2>5
 at_least_two_clear_games: score1-score2>1 or score2-score1>1
end -- class TennisSet
```

The necessary structure could be modelled by the STL-based types Set, Match and Results:

```
typedef pair<int, int> Set;

class Match : public vector<Set>
{
public:
  unsigned int wins1();
};

unsigned int Match::wins1()
{
  unsigned int wins=0;
  for (iterator p=begin(); p<end(); p++)
    wins += p->first>p->second;
  return wins;
}

typedef vector<Match> Result;
```

Clearly, the data structure is modelled, but there is no representation of the constraints. For an instance of the class Results to be valid, all the constraints of each of its components at all levels must be validated. For the Results instance {[(6-4), (4-6), (7-5), (6-1)], [(6-4), (6-2), (6-1)], [(0-6), (0-6), (1-6)]} there are fourteen such constraints.

4.1 Enhancing STL containers to monitor invariants

In order to incorporate monitoring of these invariants into the Results objects – without altering either the code of the STL, or the code shown above – the containers from which the Results class is built must inherit new behaviour, but without changing their names, and without altering their existing behaviour in any way. This can be done by using inheritance and namespaces. So, for example, given the namespace ‘stdpp’, in which the class stdpp::vector<T, Alloc> inherits from the class std::vector<T, Alloc> as follows

```
namespace stdpp {
//
template <class T, class Alloc = alloc>
class vector : public std::vector<T, Alloc>, public CoManager {
public:
 bool inv() const { return true; }
 bool determineInv() const;
//
};
} // end namespace stdpp
```

then within a namespace which uses the definition of vector<T> from the namespace ‘stdpp’, specifically with a `using` declaration, or simply within an extension of namespace stdpp. Any use of vector<T, Alloc> refers to stdpp::vector<T, Alloc>, which behaves exactly as the class std::vector<T,

Alloc>, but also has additional behaviour. If monitoring for testing purposes is all that is required, a component can be developed within a namespace which uses the validated versions of the containers, in which it can be thoroughly tested, and released in a form which, removed from the scope of the development namespace, reverts to using the standard STL containers only.

The additional behaviour comes from inheriting the class CoManager. In this simple case, the CoManager class is defined as follows:

```
class CoManager {
public:
  bool inv() const=0;
  virtual bool determineInv() const=0;
};
```

The member function ‘bool inv() const’ is for the client of the validated class to define what constitutes a valid instance of the class. Specific instantiations of the container classes within the validated namespace that inherit the CoManager class can redefine it by explicit specialisation [19]. Note the use of the `const` qualifier to try to ensure that the C++ functions used to verify assertions do not affect execution of the program. There is no way to ensure this absolutely (e.g. inv() could call abort()), but use of `const` at least tries to keep the imperative fox out of the applicative chicken coop (to borrow Meyer's delightfully graphic description)).

Note that the use of `typedef` in defining Set and Results is therefore not realistic for a non-trivial system, since the definition of Set::inv() would apply to all pairs of `int`s: derivation must be used.

The member function determineInv() is defined for each validated container: it is not defined by the container client. The purpose of the function is to determine the validity of the object stored in that container, based on:

(i) the definition of inv() provided by the client for that container itself;

(ii) knowledge of the elements of the container.

The routines simply make a recursive descent check of validity, and throw an exception when an invalid instance of a subtype is encountered.

The additional client code required to implement the constraint monitoring for the tennis example then amounts to writing the invariants:

```
bool Set::inv() const
{
  return abs(first-second)>1 && (first>5 || second>5);
}

bool Match::inv() const
{
  return wins1()==3 || wins1()==size()-3 && size()<=5;
}

bool Result::inv() const
{
  return size()<=TOURNAMENT_LENGTH;
}
```

4.2 Preconditions and postconditions

In addition to invariants on the values of a data type, specifications in languages such as VDM-SL state preconditions and postconditions on the operation of routines.

The mechanism described thus far can be extended to provide a degree of support for operation preconditions and postconditions. For containers based on the STL, the necessary extension can be provided in library form. Containers not based on the STL require additional effort.

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 preconditions and postconditions which are specific to the instantiation of an operation for a particular type. Both can be accommodated.

Each container operation for which a client may wish to define a precondition or postcondition (i.e. those that modify the container: no client is likely to want to stipulate a specific precondition for an operation such as size()) is redefined within the namespace 'stdpp' to check conditions which can be defined for specific instantiations by the client.

```
namespace stdpp {
//
template <class T, class Alloc = alloc>
class vector : public std::vector<T, Alloc>, public CoManager {
public:
 virtual bool pre_erase(iterator) { return true; }
 virtual bool post_erase(const vector<T>&, iterator){ return true; }
 iterator erase(iterator);
//
};
//
} // end namespace stdpp
```

Thus, the definition of stdpp::vector<T>::erase(iterator) becomes

```
iterator stdpp::vector<T>::erase(iterator pos)
{
  if (!size() || !pre_erase(pos))
    throw vector<T>();
  vector<T> old(*this);
  iterator newPos = std::vector<T>::erase(pos);
  if (!post_erase(old, newPos) || !inv())
    throw vector<T>();
  return newPos;
}
```

The client then has the option of supplying specific definitions of stdpp::vector<T>::pre_erase() or stdpp::vector<T>::post_erase() as necessary. Clearly granularity of control over which checks are employed can be introduced.

This section showed how a straightforward object management task – constraint monitoring – can be performed non-intrusively by taking advantage of object-oriented language features. A constraint-checking version of the STL can be adopted without changing any library or client code.

5. Extending management

In the previous section, it was demonstrated how a function to determine an object's validity could be evaluated recursively over the internal structure of an object by defining two functions: one which dealt with the structure, and one which dealt with the evaluation. The first – determineInv() – had to be defined for each type of container used (by the container-adaptation supplier), the second – inv() – had to be defined for each class (by the container client).

In the development of Config, there were a number of other services required which were concerned with ensuring the integrity of the data. These required a more sophisticated mechanism than that required for constraint checking alone. This new mechanism provides an alternative means of constraint checking to the one above. If constraint checking alone is all that is required, this new mechanism involves a lot more effort than the previous one. If that work is already done to provide other services, less effort is required to implement constraint checking.

A more comprehensive scheme than that described in the previous section was required for managing objects in Config. In this scheme, each object may be considered to be potentially self-managing, which it does by maintaining a pointer to a manager object. The make up of these managers is described below, and the uses to which they were put in the Config system are documented. One clear advantage of

creating separate manager classes in this way is that the managers can be given a uniform structure. This eliminates the need to specialise constraint checking functions such as determineInv() for each type of container, reducing the work for the container-adaptation supplier.

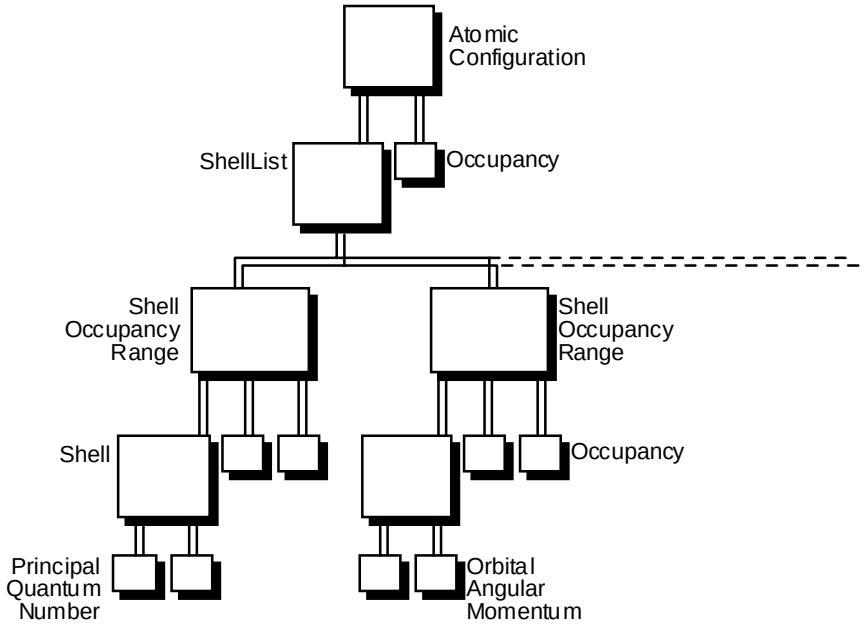

Figure 4. Atomic Configuration shadowed by Managers.

Consider the type AtomicConfiguration. This is the type entered by the user, from which a ConfigurationTreeList is constructed. In order to monitor the validity of this data structure, it is shadowed by a structure of managers (shown black in Figure 4).

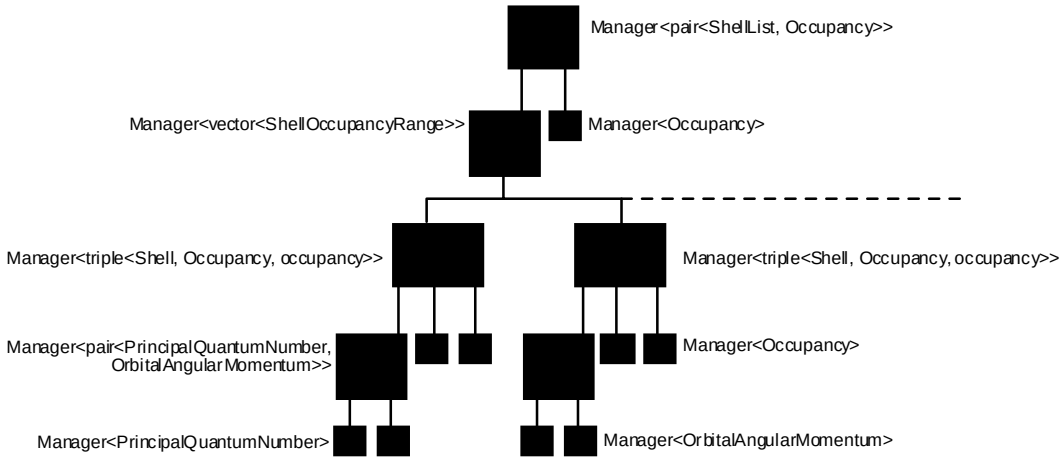

Figure 5. Managers for an AtomicConfiguration.

The Managers are all instantiations of the same generic class, Manager<T>. Use of partial specialisation means that it is possible to write generic managers for handling each container type. Thus, the Managers for an AtomicConfiguration and a Shell are both instantitations of the same partial specialisation Manager<pair<T1, T2>>. The Manager structure is shown in Figure 5.

5.1 The classes involved

The scheme is based on a small class hierarchy used to define the generic *Manager* class, and a class with which instantiations of Manager interact, CoManager. This time, CoManager is a generic class, which takes the type of the container it is inheriting as a parameter. This can look like a circular definition at first sight.

```
template <class T, class Alloc = alloc>
class vector : public CoManager<vector<T, Alloc>>, public std::vector<T,
Alloc> { // };
```

CoManager<C<T>> defines a member newManager(..), and each enhanced container's constructor calls newManager(..) recursively to create the Managers of its elements. Thus declaring a Manager for an AtomicConfiguration creates an entire Manager hierarchy (Figure 5).

Although there are different Manager classes – one for each intrinsic type and one for each container class – each Manager object has a common ancestry. Most of the functionality of these Managers will be shown to be independent of the type that instantiates them, or automatically derivable from it. Consequently, there is little effort involved in writing Managers for non-STL classes.

At run-time, the Manager objects form a hierarchy which parallels the internal data structure of a managed object. This does not contravene Meyer's principle of Knowledge Distribution since the Manager hierarchy is generated automatically from the data structure.

Each Manager performs tasks appropriate to its position in the hierarchy, concerning itself only with the type that instantiated it, not with its components or with its parent. There are three layers of classes, which progressively diversify. The first layer, a single, non-generic class, is shared (inherited) by all Managers. The second layer is parameterised by the type it is validating, but the code is shared. The final layer is again essentially parameterised by the type it is validating, and is specialised for each basic type and container type.

5.2 Uses of Manager<c<T>>

Thus far a mechanism for automatically generating a manager structure that parallels the data structure of an object has been described, and mention has been made of how it can be used to monitor the validity of the state of, and operations on, an object.

This section shows the uses to which the mechanism has been put in the Config system (outside of that realm the uses are far from fully explored).

The first use is in 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 an OrbitalAngularMomentum (Figure 3).

The scheme is that where appropriate Managers provide operations that change 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. This process of assembly propagates up the structure through the hierarchy of Managers.

In summary, 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.

In addition, validity is checked as the call progresses up the hierarchy. constructValues(..) returns the validity of the value it has constructed. This can be much more economical than a recursive descent check

from the top. Only those parts of the hierarchy affected by the update need to have their validity re-evaluated.

The second use is in determining the availability of edit operations. Considering again Figure 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 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.

The third use is in data display prototyping. One of the aims of the G*R*ACE 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 [20] 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 [21] 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.

5.3 Limitations

Encapsulating built-in types is an issue. Function definitions such as constructValues(..) are necessarily going to rely on recursive decompositions which take advantage of the characteristics inherited by the container in the validated namespace; however, the lowest-level containers will have elements of built-in type, which cannot be made to exhibit the additional behaviour. It is therefore necessary to write partial or explicit specialisations for each container that override the general definition for cases where a built-in type is a type parameter, or to require that all built-in types are encapsulated into a class before use.

Kick-starting manager generation is an issue. Most classes in the Config system are created by deriving from STL instantiations, in order to add functionality or to rename to better reflect the terminology of the problem domain.

In order for the mechanism to work in such cases, it is vital, for example, that the function

```
bool pair<ShellList, Occupancy>::inv() const;
```

is defined and not AtomicConfiguration::inv() const (contrast Figure 4 and Figure 5).

This is because the compiler does not perform type conversions on the type parameters of a specialisation (and so no definition of Manager<AtomicConfiguration> would be available to the

compiler). This means that when a Manager needs to be declared, it is necessary to use the container instantiation, not its derivative.

Implementing implicit specification is an issue. The generation of configuration trees from an atomic configuration is specified by list inclusion. The corresponding validity check can show that all trees generated are valid, but cannot show that all valid trees are generated. The monitoring as it stands will not detect this: if the algorithm produces a result which meets a stronger condition than the one given in the specification, the result will be passed as valid (which it is).

6. Conclusion

Implementers of software components who adopt an object-oriented architecture have to address a number of concerns with respect to the management of objects. In C++, object containers and algorithms that act on them are provided by the STL, but facilities to deal with other concerns are not: monitoring of class invariants and operation preconditions and postconditions must be added, as must display of the object data and mechanisms for control and execution of object update.

It has been shown that all of these can be provided, in a non-intrusive fashion, for any types based on the containers of the STL. It is demonstrated how key features of the object-oriented paradigm such as inheritance and virtual functions, along with other language features such as generic programming and namespaces, make it possible to do this in such a way that practically all code is parameterised on the type of the object, code duplication is all but eliminated, and type-specific code is only written when absolutely necessary. The simplest implementation of constraint monitoring can be adopted for existing STL-based code simply by providing an encoding of the constraints to be monitored.

1 David Maley, Ivor Spence. Config: A G*R*ACE tool for generating configuration trees. CPC 114 (1998). Elsevier.
2. V.M. Burke, P.G. Burke, N.S. Scott. Computer Physics Communications 69 (1992). Elsevier, North Holland.
3. Cliff Jones, Systematic Software Development using VDM. 2nd Edition. Prentice Hall. 1990.
4. Scott, Kilpatrick, Maley. The Formal Specification of Abstract Data Types and their Implementation in Fortran 90. Computer Physics Communications 84 (1994) 201-225. Elsevier, North Holland.
5. Eugène Dürr. The Use of Object-Oriented Specification in Physics. Ph.D. Thesis, University of Utrecht. 1994.
6. Grady Booch. Object Oriented Analysis and Design, 2nd Edition. Addison Wesley, 1994.
7. Peter Coad and Edward Nash Yourdon. Object-Oriented Analysis, Prentice Hall, 1990.
8. Betrand Meyer. Object-Oriented Software Construction, 2nd Edition. Prentice Hall, 1997.
9. Bertrand Meyer. Applying Design by Contract. Computer (IEEE) vol 25 no. 10 p40-51 October 1992.
10. Alexaner Stepanov (SGI) & Meng Lee (HP Labs). The Standard Template Library. 1995.
11. Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841. November 1994.
12. http://www.stmarys-belfast.ac.uk/acad/depts/cit/dm
13. Robert V. Binder. Testing Object-Oriented Systems: Models, Patterns and Tools. Addison Wesley, 1999.
14. David Welch & Scott Strong. An Exception-based Assertion Mechanism for C++. JOOP, July/August 1998.
15. Erich Gamma et. al. Design Patterns for Object Oriented Software. Addison Wesley. 1994.
16. David Rosenblum. A Practical Approach to Programming with Assertions. IEEE Transactions on Software Engineering, Vol 21, No. 1, Jan 1995.
17. Sara Porat & Paul Fertig. Class Assertions in C++. JOOP, May 1995.
18. Maurice Cline and Doug Lea. Using Annotated C++. Proceedings of C++ at Work. September 1990.
19. C++ International Standard ISO/IEC 14882, Section 14.7.3. ANSI 1998.
20. Marshall Brain. MOTIF - The Essentials and more. Digital Press. 1992.
21. Douglas Young. Object-Oriented Programming with C++ and OSF/MOTIF. Prentice Hall, 1995.