Contract-oriented specifications

Richard Mitchell, John Howse and Ali Hamie

Division of Computing
University of Brighton
Brighton BN2 4GJ, UK

{Richard.Mitchell, John.Howse, A.A.Hamie}@brighton.ac.uk

Abstract
In classes developed using design-by-contract, contracts contain assertions that formalise preconditions, postconditions and invariants. To be sure that contracts are complete, they can be derived from specifications. For classes in a data structures library, equational specifications are appropriate. However, a conventional equational specification cannot usually be mapped directly to contracts. Instead, a second, contract-oriented, equational specification can be devised, with two key properties: it can be proved that the contract-oriented specification implies the original specification; and the contract-oriented specification can be mapped systematically to contracts. These two properties combine to increase confidence that the contracts capture the same abstraction as the equational specification.

1 Introduction

Contracts between the suppliers of services provided by objects and the clients of those services contain assertions expressing preconditions and postconditions on individual services, and invariants on complete types or classes. Contracts can be expressed formally or informally, and can be used in analysis and design models as well as in programs (see Figure 1). Fusion (Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes and Jeremaes 1994) uses pre and postconditions to define operations, and this approach has been carried over to the Unified Modelling Language (Booch, Jacobson and Rumbaugh 1997). Syntropy (Cook and Daniels 1994) and Catalysis (D'Souza and Wills 1997) provide a formal notation for expressing contracts.

At the programming level, Eiffel (Meyer 1992) provides direct support for contracts by providing

- placeholders for preconditions, postconditions and invariants
- special operators (particularly the **old** operator) to support the writing of contracts
- automatic checking of assertions (with programmer control over which assertions are checked).

In programming languages other than Eiffel, informal contracts can be expressed in the form of comments, and formal contracts can be simulated by using macros or functions for defining assertions, and by making explicit calls to evaluate them.

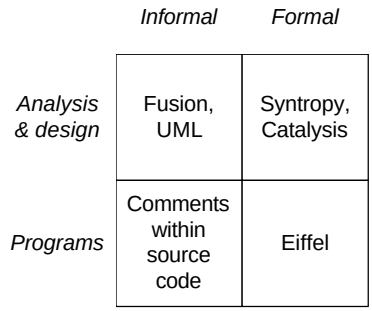

Figure 1. *Examples of where contracts can be found*

Within Eiffel programs, formal contracts in class interfaces can be expressed using boolean expressions written in Eiffel. This means they can be executed, to check that objects of the class behave according to the stated contract.

The work reported here is part of a larger programme to explore how the use of mathematically-grounded ideas can be used within object technology. This paper addresses the question of how to move from an equational specification of a type to contracts within a corresponding class, with

the competing goals of being confident that the contracts are complete and are faithful to the original specification yet without being fully formal at every step. The basis of the answer offered here is to devise a second formal specification to support the journey from a given formal specification to contracts at the program level. This approach builds on ideas in (Meyer 1994a) which proposes adding features to a class in order specifically to be able to write richer, more expressive contracts.

Our work is currently focused on classes from a data structures library (Meyer 1994b), so equational specifications make suitable starting points. We have chosen the Larch specification language (Guttag and Horning 1993) because it has a handbook of specifications that have been exposed to public scrutiny, and an associated theorem prover (Garland and Guttag 1991).

We have carried out most of our design-by-contract experiments using Eiffel, because it provides runtime checking of assertions, making the experiments easy to conduct. For the sake of clarity, we do not use the full power of the Eiffel language in the examples (for instance, we have avoided anchored declarations). A brief explanation of relevant Eiffel-specific terminology is given in Appendix A.

The paper is organised as follows. The next section introduces three styles of specification. "Property-oriented" specifications concentrate on defining the properties of data types without discussing computational issues such as how to arrive at results. "Construction-oriented" specifications include algorithmic information on how to arrive at results. Finally, "contract-oriented" specifications, introduced in this paper, can be mapped systematically to contracts in programs and can be connected to corresponding property-oriented specifications. Contract-oriented specifications thus form a stepping-stone between conventional property-oriented specifications and contracts in programs, allowing us to have confidence that our contracts describe the same abstraction as the corresponding property-oriented specifications, which we take as the starting point for specifying a class interface.

Section 3 presents more details of the route from property-oriented specification to program-level contracts. Figure 2 shows an overview of the route. Section 4 discusses how the route supports the claim that the contracts capture the same abstraction as the property-oriented specification. In practice, class interfaces contain commands—features that change the state of the receiving object. Section 5 shows how contracts on functions are connected to contracts on commands, and includes a brief discussion on whether encapsulation is compromised by the addition of features to support contracts. Section 6 reviews what has been achieved and looks forward to work that still needs doing.

We have used the data type SET as the basis for our examples. In our experience, it is the trickiest to work with of all the simple data types, because sets have no internal structure. If you understand the way we handle sets, you will be able to work out how to handle stacks, queues, lists, binary search trees, and the like.

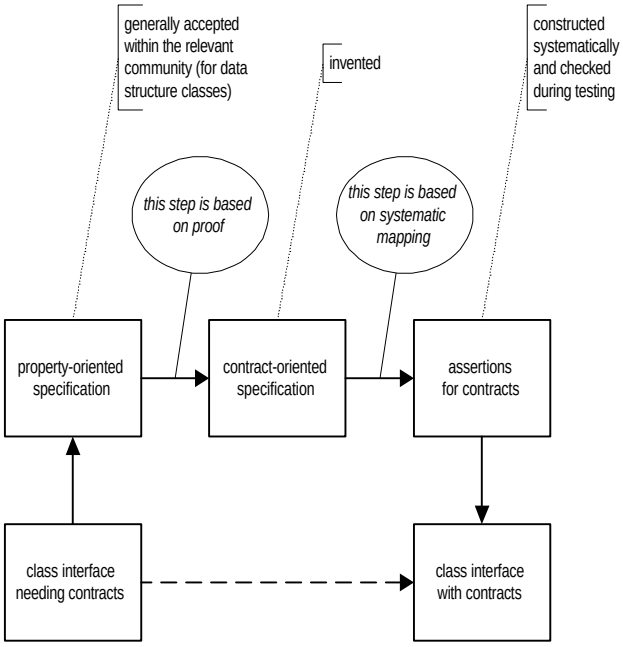

Figure 2. An overview of the route

A class interface needing contracts is turned into one with contracts (dotted arrow) via this route: an appropriate property-oriented specification of the underlying abstract data type is mapped to a contract-oriented specification and then to assertions that can be inserted as contracts; the mapping is supported partly by proof and partly by being systematic. For data structure classes, the property-oriented specifications are usually already available. For other domains, they would need devising and agreeing

We have not tried to measure the effort it takes to write contracts in our style, or any other style, and we do not know how to measure the savings that might result from using contracts. Therefore, we do not claim any cost-benefit for our approach. For our own peace of mind, we assume that there could be software development projects that would benefit

from the very careful use of contracts (by which we mean doing better than devising contracts in an ad hoc manner, but stopping short of full, formal reification of specification to code). For instance, one such project might be to build, test and document a data structure library that is to serve as the basis of an international standardisation effort.

2 Three styles of specification

Here is part of a Larch specification of the type SET, showing the signatures of some of SET's functions. SET is generic in the type, G, of elements it contains.

```
SET( G )
   null : -> Set
   plus : Set, G -> Set
   has : Set, G -> Bool
   union : Set, Set -> Set
```

The specification continues by listing a number of axioms that define the properties of these functions.

```
∀ s, s1, s2 : Set, g, g1, g2 : G
   ¬ has( null, g )
   has( plus(s, g2), g1 ) ==
          g2 = g1  ∨  has(s, g1)
   has( union(s1, s2), g ) ==
          has(s1, g) ∨ has(s2, g)
```

The first two axioms capture the relationship between the function 'has', which accesses a set value to produce a boolean value, and the two functions 'null' and 'plus' that, between them, can be used to generate every possible set value; these two axioms thus define 'has'. The third axiom uses 'has' to define 'union'. The style of this axiom is property-oriented (Jones 1986). It asserts the property of the union of sets 's1' and 's2' that, for any element, the union has the element if, and only if, either of the sets used to form the union has that element. But it gives no clue about how to construct the union of 's1' and 's2'.

Here is another way to specify 'union', in what we call a construction-oriented style.

```
∀ s1, s2 : Set, g : G
   union( s1, null ) == s1
   union( s1, plus(s2, g) ) ==
        union( plus(s1, g), s2 )
```

These two axioms could readily be turned into the definition of an executable function in a functional programming language. The axioms show how this function can construct its result, which is why we say that the corresponding specification is construction-oriented.

Now we turn to the world of object-oriented programs. Here is the declaration of a feature 'union' within the interface to a class SET.

```
class interface SET[G]
…
   feature
     union(other : SET[G]) : SET[G]
-- A new set object holding the union
-- of 'Current' and 'other' and sharing
-- the elements of 'Current' and
-- 'other'
        ensure
           …
```

The keyword 'ensure' introduces a postcondition that will define the result of invoking 'union'. Sadly, this postcondition cannot be derived directly from either of the two specifications we already have. We want to write a postcondition that asserts what is true about the result of invoking 'union', so the property-oriented specification has the right flavour. The two sets known as 's1' and 's2' in the relevant axiom are known as 'Current' and 'other' in the program, but the element 'g' in the axiom has no counterpart in the program.

The construction-oriented specification is geared to a computational world—it defines an algorithm for reaching the required result. Can we work from that specification? It, too, is not suitable for our purposes, because it, too, refers to an element 'g' within the pattern-matching expression on the left-hand side of the axiom.

This leads us to a third style of specification, which we call contract-oriented because specifications in this style can be mapped in a straightforward manner to contracts in programs. Here is the specification of 'union' in the contract-oriented style.

```
union( s1, s2 ) ==
   if size(s2) = 0 then s1
   else
      union(
        plus( s1, last_added(s2) ),
        except_last_added(s2) )
```

This will have been preceded by declarations of two new functions:

```
last_added : Set -> G
except_last_added : Set -> Set
```

We call functions like these "deconstructors" (Mitchell 1996, Mitchell and McKim 1996), because they are used to reveal the structure of a

value of type set. Sets are generated ("constructed", in a mathematical sense) using 'null' and 'plus'. As a result, some axioms have a left-hand-side of the form

```
f( plus(s, g) ) ==
```

in which the pattern 'plus(s, g)' is used to uncover the structure of an un-named set that is being used as an argument to some function 'f'. The right-hand-side of the axiom can then refer to the component parts, 's' and 'g'. There is no pattern-matching mechanism in conventional object-oriented programming languages, so we add features, whenever necessary, to achieve the same effect. In the case of sets, the features we need are 'last_added' and 'except_last_added', and they are defined by these axioms:

```
last_added( plus(s, g) ) == g
except_last_added(plus(s,g)) == s
```

There is a discussion later on the propriety of adding such features. For now we mention that the full contract-oriented specification of SET also contains a set equality function 'is_equal' that ignores different element orderings exposed by 'last_added' and 'except_last_added'.

The first step in mapping a contract-oriented specification to contracts in a class is to map each function in the specification to a feature in the class, and then to work on individual functions. Continuing with 'union' as the example, we break its axiom into two, based on the axiom's if-then-else structure, and introduce 'is_equal' wherever equality of sets is involved.

```
size(s2) = 0 ⇒
     is_equal( union(s1, s2), s1 )

not(size(s2) = 0) ⇒
  is_equal(union(s1,s2),
     union(plus(s1,last_added(s2)),
            except_last_added(s2)))
```

For each of the new axioms, we choose a feature whose postcondition can capture the axiom. Both the above axioms can be captured in postconditions on 'union'. (The deployment of axioms to postconditions is not always so direct. The axiom for 'last_added', for instance, becomes a postcondition on the feature 'plus'. The axioms for 'size' are spread between the features 'null' and 'plus'.) Next, we map the elements of the axioms to elements of the program. To do this, it is helpful to see what the axioms look like in an object-oriented notation, in which one argument to each function has been distinguished as the "receiver".

```
s2.size = 0 implies
       s1.union(s2).is_equal(s1)

not (s2.size = 0) implies
  s1.union(s2).is_equal(
     s1.plus(s2.last_added).union(
            s2.except_last_added))
```

The mapping to elements of the program is as follows.

```
s1 maps to Current (the receiver)
s2 maps to other (the argument)
union(s2) maps to Result
```

Putting all the steps together, we now have a feature 'union' with a postcondition that is derived systematically from a specification.

```
  union( other : SET[G] ) : SET[ G ]
-- A new set object holding the union
-- of 'Current' and 'other' and
-- sharing the elements of 'Current'
-- and 'other'
    require
      other_not_Void:
        other /= Void
    ensure
      other_is_empty_means_Result_
            equals_Current:
        other.size = 0 implies
            Result.is_equal(Current)
      other_not_empty_means_Result_
          defined_recursively:
        not (other.size = 0) implies
          Result.is_equal(plus(
          other.last_added).union(
          other.except_last_added))
```

(A typical precondition has been added, introduced by the keyword **require**, to make sure that the argument 'other' actually does refer to some object.)

3 A second example

This second example shows how we usually present the mapping from specification to postconditions. Earlier, we introduced these two property-oriented axioms for 'has':

```
¬ has( null, g )
has(plus(s,g2),g1) ==
              g2 = g1 ∨ has(s,g1)
```

and observed that we could not systematically map from these axioms to postconditions on features of a

SET class. Here is an axiom for 'has' in the contract-oriented style that we *can* map:

```
has(s, g) ==
  if size(s) = 0 then false
  else
    if last_added(s) = g then true
    else
      has(except_last_added(s),g)
```

This can be split into its 3 cases (the change from "==" to "=" and the logically unnecessary use of "= true" are immaterial Larch details).

```
size(s) = 0 ⇒ has(s, g) = false
¬(size(s) = 0) ⇒ (last_added(s) = g
          ⇒ has(s,g) = true) = true
¬(size(s) = 0) ⇒
  (¬(last_added(s) = g) ⇒
    has(s,g) =
      has(except_last_added(s),g))
```

Because the original axiom has a nested if-then-else structure, the second and third of these axioms have complex guards, of the form

```
guard1 ⇒ ( guard2 ⇒ definition ).
```

Such guards are mapped to an object-oriented form using a conditional 'and' operator (spelt **and then** in Eiffel), like this:

```
guard1 and then guard2 implies
  definition.
```

The three axioms can now be mapped, one by one, to postconditions on features in class SET. We show the mappings in the form in which we normally present them.

has - axiom 1

Definition:

```
size(s) = 0 => has(s,g) = false
```

In OO syntax:

```
s.size = 0 implies s.has(g) = false
```

Postcondition on:

```
has
```

Map:

```
s to Current, has(g) to Result
```

Assertion:

```
size = 0 implies Result = false
```

has - axiom 2

Definition:

```
¬(size(s) = 0 ) ⇒ ( last_added(s) = g  ⇒ has(s, g))
```

In OO syntax:

```
not (s.size = 0) and then
  s.last_added = g implies s.has(g)
```

Postcondition on:

```
has
```

Map:

```
s to Current, has(g) to Result
```

Assertion:

```
not ( size = 0 ) and then
  last_added = g implies Result
```

has - axiom 3

Definition:

```
¬(size(s) = 0) ⇒
  (¬(last_added(s) = g) ⇒
    has(s,g) =
      has(except_last_added(s),g))
```

In OO syntax:

```
not (s.size = 0) and then
      not s.last_added = g
  implies s.has(g) =
    s.except_last_added.has(g)
```

Postcondition on:

```
has
```

Map:

```
s to Current, has(g) to Result
```

Assertion:

```
not ( size = 0 ) and then
      not( last_added = g )
  implies
  Result = except_last_added.has(g)
```

The feature 'has' in the interface of class SET looks like this.

```
has( g : G ) : BOOLEAN
  -- Is 'g' an element of the set?
  ensure
    set_is_empty_means_result_
        must_be_false:
      size = 0 implies Result = false
    looking_for_last_element_
        added_means_result_is_true:
      not (size = 0) and then
      last_added = g implies Result
    otherwise_recur:
      not (size = 0) and then
      not (last_added = g) implies
    Result = except_last_added.has(g)
```

It is more important that this resulting postcondition, and the property-oriented specification it is derived from, should be readable, than that the intermediate forms should be readable.

The specification and the contracts form the basis of communication between developers of classes and users of those classes. The intermediate forms, including the contract-oriented specification, are the province of just the developers.

4 Validating and verifying the contracts

How can we be sure that the postcondition just derived for 'has' is capturing what we originally specified about 'has' for sets? The answer to this question is in several parts, reflecting the various steps from class interface to contracts shown in Figure 2.

We begin by assuming that a property-oriented specification is validated within the community that needs it (for example, by agreement between developer and customer authority, or by general acceptance of published work). In our own work, to help to avoid mistakes and unintended bias, we always start from published specifications (except our own!).

The next part concerns how we show that a contract-oriented specification is consistent with a property-oriented one. For example, given a property-oriented specification of SET, containing axioms for 'has' such as these:

```
¬has(null, g)
has(plus(s,g2),g1) ==
                  g2 = g1 ∨ has(s,g1)
```

how can we be sure that a contract-oriented version containing the following three-part axiom says the same?

```
has(s, g) ==
  if size(s) = 0 then false
  else
    if last_added(s) = g then true
    else
      has(except_last_added(s),g)
```

The answer is that we prove that the contract-oriented version implies the original property-oriented version. Here is an outline of the key steps in part of the proof. The theorem to be proved is the second axiom in the property-oriented specification of 'has'. The proof starts from the contract-oriented specification of 'has' and establishes the property-oriented axiom as a theorem. Proving this theorem is equivalent to showing that any program that satisfies the contract-oriented specification will also satisfy the property-oriented one, too. The proof and its supporting lemmas have been checked using the Larch theorem-prover.

Theorem

```
∀ g1, g2 : G, s : Set
  has(plus(s, g2), g1) ==
                g2 = g1 ∨ has(s, g1)
```

Proof

```
∀ g1, g2 : G, s : Set
  has(plus(s,g2),g1) ==
    if size(plus(s,g2)) = 0 then
      false
    else
      if last_added(plus(s,g2)) = g1
        then true
      else has(except_last_added(
        plus(s,g2)),g1)
```

(instantiate the contract-oriented axiom for has with s replaced by plus(s,g2))

```
    == if g2 = g1 then true else
         has(s, g1)
```

(size(plus(s,g)) ¹ 0 by lemma, last_added(plus(s, g2)) = g2 by definition of last_added, except_last_added(plus(s, g2)) = s by definition of except_last_added)

```
    == g2 = g1 ∨ has(s, g1)
```

(by definition of if-then)

Devising the contract-oriented version involves invention, by someone who understands the property-oriented version, and who knows what can and cannot be mapped to contracts.

Once a contract-oriented specification has been devised and verified, it can be mapped systematically to contracts (mostly, postconditions), without the need for much invention along the route (the person devising the mapping must choose which feature's postcondition an axiom maps to, but getting this wrong makes it impossible to carry out the mapping). The mapping is systematic, but there is no proof that the postconditions follow from the specification. The benefit of this is that it is not necessary to construct proofs in a logic rich enough to encompass both specifications and programs. The cost is that we rely on being systematic to verify the mapping from specification to contracts (we believe we can construct a program to check the mapping, but we have not done so yet).

The contracts are subject to further validation. As a result of the way they have been derived, the postconditions are executable. As the program is tested, the postconditions are evaluated. Should any

of the postconditions evaluate to false on any test, the tester will be alerted. People can then judge whether it is the implementation or the postcondition that is faulty. Confidence in the postconditions is increased by the fact that they must survive the testing process.

5 Relationship to commands

The features examined so far, 'union' and 'has', were both mapped to functions at the program level. Usually, a class will contain commands to modify the receiving object. Imagine that a class SET contains a command to perform a union-like operation:

```
add_in( other : SET[G] )
   -- Modify 'Current' to be the
   -- union of 'Current' and 'other'
```

The feature 'add_in' can have a contract that relates it to 'union' like this:

```
add_in( other : SET[G] )
   -- Modify 'Current' to be the
   -- union of 'Current' and 'other'
   require
      other_not_Void:
         other /= Void
   ensure
      effect_defined_by_union:
         is_equal(old union(other))
```

The postcondition has elided references to the current, or receiving, object; in full it reads

```
Current.is_equal(
       old Current.union(other))
```

If a feature such as 'add_in' is to be defined, and there is not a corresponding function such as 'union', our approach is to introduce one. This means that class interfaces are larger than usual. (We assume compilers can remove unused code, so that compiled production programs are not necessarily larger. For instance, if postcondition checking is turned off, features introduced only to write postconditions could be omitted by a compiler.) In return for reading a larger class interface, the programmer gets contracts that are derived from specifications. To show the kinds of features that are added, here is what a STACK class might contain. The features have been loosely categorised.

```
-- creation feature
   make
-- commands
   put( g : G )
   delete
-- queries
   top : G
   size : INTEGER
   is_equal(other: STACK[G]) : BOOLEAN
-- additional functions
   plus( g : G ): STACK[ G ]
   minus_top: STACK[ G ]
```

The features 'plus' and 'minus_top' are the functions corresponding to the commands 'put' (often called "push", but we have adopted the uniform naming convention advocated by Meyer (1994b)) and 'delete'. These functions can be given contracts derived from a specification. The commands can then be defined in terms of the functions.

A note on encapsulation

A function such as 'minus_top' added to a STACK class reveals nothing to a client of the class that could not be determined using the regular features such as 'delete' and 'top'. Even a feature that reveals the element at any position in a stack does not break encapsulation. A feature such as

```
element_at( i : INTEGER ) : G
-- The element at position 'i' on
-- the stack, where the bottom is
-- position one and the top is
-- position 'size'.
```

can readily be programmed by a client, using only normal client features such as 'size', 'top' and 'delete'. The implementation might be very slow, and might destroy the receiver, but these are not the issues.

The situation is a little different with some data types, and SET is the trickiest we have encountered so far. The examples have introduced two features added to SET to support the contract-oriented specification approach:

```
last_added : G
-- The most recently added element
-- that has not been removed
except_last_added : SET[G]
-- A new set formed by discarding
-- the most recently added element
```

If I am a client that builds a set object, I know what was the last element added (and all the ones before that). If, however, I am a client that is given an existing set, I cannot know what was the last element added—sets do not have enough mathematical structure to support such a notion.

We distinguish, therefore, between features that are knowable only by a client that builds an object from those that are knowable by a client that

dismantles an object (these will also be knowable by a builder client). Our goal is only to add features to a class that are knowable by a dismantler. The example of 'element_at' for class STACK presented earlier would be one such feature. It is an unusual feature to find in the interface to STACK, but actually reveals no implementation details that a client prepared to dismantle a stack object could not find out using the regular features.

For SET (and data structures such as DICTIONARY that have underlying set-like properties), we cannot quite attain the goal, and we are forced to add features that give some logical internal structure to sets. However, features such as 'last_added' do not reveal anything about the chosen implementation. That can still be changed without any impact on the class interface. In addition, the features mimic the structure given to sets in the formal specification world, which uses pattern-matching of the form 'plus(s, g)' to reveal 'g', the 'last_added' element, and 's', the set 'except_last_added'. Finally, recall that both the specification and the implementation of SET contain a set equality function 'is_equal' that ignores different element orderings exposed by 'last_added' and 'except_last_added'.

A note on interfaces

Although the interface to class SET must contain the features 'last_added' and 'except_last_added' so that they can appear in contracts, there is no reason why a project should not adopt the convention that such features are used in test programs but not in normal clients. With a little help from tool builders, it would be possible to define that such features are visible only to the assertion-checking mechanism and to human readers, thus enforcing the convention.

In practice, we are not nervous about writing clients that use added features of the kind that reveal nothing that could not already be determined by a client prepared to dismantle an object (for example, 'minus_top' on a stack). We are nervous about writing clients that use added features that reveal information that only a builder of the object could know (for example, 'last_added' on a set). We add such features reluctantly to server classes, we avoid calling them from client classes, and we look forward to tools that enforce these self-imposed restraints.

6 Conclusions and continuing work

The preceding sections have shown how contracts in a class implementing a simple data structure can be derived from a generally-accepted equational specification. The derivation involves proof in the first stage and systematic mapping in the second stage. Because the first stage uses proof and the second stage involves no change of level (i.e., no reification), there is no need for an abstraction function usually associated with refinement from specification to program.

Related work includes Larch interface languages for Smalltalk (Cheon and Leavens 1997) and C++ (Leavens 1997) and the formal specification of generic components of the C++ Standard Template Library (Musser 1997). However, none of this work considers mapping equational specifications into machine checkable assertions.

There are several strands to our continuing work in the area of contracts. First, we are defining contracts for further types of data. We are working on those to be found in data structure libraries, and we are beginning to look at how to write contracts for application-oriented classes, which are usually not so easy to treat "stand-alone" as data structures.

Some contracts are, perhaps, easier to write and read if they are expressed using quantified assertions. The Eiffel libraries (Meyer 1994b) make use of quantified assertions, inserted as comments. McKim (1996) advocates inserting quantified assertions as comments as a means of achieving complete descriptions of class interfaces. We have a prototype of an approach to supporting runtime-checkable, universally and existentially-quantified assertions in current Eiffel. The programmer must, again, define additional functions in the class. The core of the approach is to keep a list of all objects that have ever been created during one execution of a program, and to quantify over that list. This contrasts with an approach based on iterating through collections (Katrib and Coira 1995); there, it is not possible to assert properties of objects not in the collection, such as that all the objects that were not in the collection are still not. We look forward to object-oriented programming languages that give more support for quantified assertions.

The contracts derived from property-oriented specifications, via contract-oriented specifications, are complete when seen as abstract specifications. From a run-time checking perspective, however, they are incomplete. Here is an example of incompleteness. Some postconditions are recursive

in form. For example, the postcondition on SET 'union' contains the assertion

```
Result.is_equal(
  plus(other.last_added).union(
    other.except_last_added))
```

which compares 'Result' (the union of 'Current' and 'other') with another expression involving 'union'. The evaluation of this postcondition is not recursive. The reference to 'union' in the second line invokes the 'union' feature, and causes the execution of the body of the feature. As a result, it is possible to devise bugs in the body of the feature that remain undetected by the postcondition. Such bugs must exploit the fact that the results of two executions of the body are compared for equality by the postcondition, and so both can be wrong, provided they are wrong "in the same way". We have a prototype of an approach to plug this hole, based on executing the recursive specification from which the contract was derived, and checking that the body of the feature delivers a matching result.

In this paper, we have simplified the testing for equality between two container objects by pretending that one 'is_equal' will do. In fact, at least two equality tests are needed, one that tests that two containers contain the same objects, and another that tests that corresponding objects in two containers have the same contents. (Equivalently, containers can be commanded to test equality using one or other condition.) We routinely include tests for 'contains_same_objects_as' and 'contains_objects_with_same_values_as', and carry the testing of both these aspects of equality of containers recursively down to the level of values.

Finally, in our work on contracts, it is our goal to develop different levels of checking, to provide different levels of "certification" of software, each with its own costs and benefits.

Acknowledgements

We gratefully acknowledge the support of the UK EPSRC under grant number GR/K67304. We thank Franco Civello, Stuart Kent and John Taylor for their helpful comments on an earlier draft.

Appendix A: Some Eiffel terminology

assertion A boolean expression used to document a program whose falsehood signals a fault in the program. An assertion can be presented as a sequence of boolean expressions, each of which can be preceded by a tag (a label). The complete assertion is the logical 'and' of the individual boolean expressions.

attribute A data item; clients have read-only access to exported attributes.

class interface The type of a class—the signatures of its visible features and a definition of their behaviour via comments and contracts.

creation When an object is created it is initialised by one of the designated creation routines.

command A routine that might change the state of the receiving and other objects but which does not return a result.

Current Eiffel's term for *self* or *this*. When Current is the receiver, it need not be stated.

ensure Keyword that marks the beginning of a postcondition.

feature A routine or an attribute.

is_equal The function *is_equal* tests object contents for equality (contrast this with the operator "=", which tests object identities for equality).

old A term of the form **old** *expression* can appear in postconditions, and refers to the value of the expression before the method began to execute.

precondition An assertion that defines the conditions under which it is valid to call a routine.

postcondition An assertion that defines the outcome of a method. If the method is a command, the postcondition defines the effect of the command by defining the relationship between the values of queries after the method has executed and their values before the method executed. If the

method is a query, the postcondition defines the result.

query — A routine that returns a result and does not change the (logical, visible) state of any objects, or an attribute—exported attributes are read-only accessible by clients.

require — Keyword that introduces a precondition.

Result — A predefined variable to hold the value to be returned by a function.

routine — A procedure, which carries out a command, or a function, which carries out a query. All routines are defined within some class.

tag — A label preceding a boolean expression within a contract. Tags are not necessary, but they improve documentation and are useful debugging aids—when an assertion fails its tag is amongst the pieces of information presented to the programmer, and helps pinpoint the cause of the failure.

References

Booch G, Jacobson I and Rumbaugh J (1997). *The Unified Modelling Language for object-oriented development.* [Online]. Available: Rational Software Corp. http://www.rational.com/ot/uml/1.0/index.html

Cheon Y and Leavens G (1994): The Larch/Smalltalk Interface Specification Language. ACM Transactions on Software Engineering and Methodology, 3(3):221-253, July 1994.

Coleman D, Arnold P, Bodoff S, Dollin C, Gilchrist H, Hayes F and Jeremaes P (1994). *Object-oriented development. The Fusion method*. Prentice Hall.

Cook S and Daniels J (1994). *Designing object systems*. Prentice Hall.

D'Souza D and Wills A (1997). *Component-based development using Catalysis*. [Online]. Available: ICON Computing Inc., http://www.iconcomp.com

Garland S and Guttag J (1991). *A guide to LP, the Larch Prover*. Technical Report TR82. Digital Equipment Corporation, Systems Research Center.

Guttag J and Horning J (1993). *Larch: languages and tools for formal specification.* Springer-Verlag.

Jones C B (1986). *Systematic software development using VDM*. Prentice-Hall.

Katrib M and Coira J (1995). *Improving Eiffel assertions using quantified iterators (draft).* Available: University of Havana, <mkm@matcom.uh.co>

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

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

Meyer B (1992). *Eiffel. The language*. Prentice Hall.

Meyer, B (1994a). *Beyond design by contract*. TOOLS Pacific '94 Keynote slides. ISE

Meyer B (1994b). *Reusable software. The base object-oriented component libraries.* Prentice Hall.

Mitchell R and McKim J (1996). *Design by contract*. TOOLS USA '96 Tutorial. ISE

Mitchell R (1996). *Software contracting using deconstructors*. Technical Report UBC 96/01. University of Brighton.

Musser D. (1997) *The Standard Template Library*. [Online]. Available at: http://www.cs.rpi.edu/~musser/stl.html