Abstract

We use an algebraic technique of formally specifying a module that implements an abstract data type, with a C++ implementation. An explicit mapping from implementation states to abstract values is added to the C++ code. The form of specification allows mechanical checking of desirable properties such as consistency and completeness, particularly when operations are added incrementally to the data type. During unit testing, the specification serves as a test oracle. Any variance between computed and specified values is automatically detected. When the module is made part of some application, the checking can be removed, or may remain in place for further validating the implementation. The specification, executed by rewriting, can be thought of as itself an implementation with maximum design diversity, and the validation as a form of multiversion-programming comparison.

1 Introduction

Encapsulated data abstractions, also called abstract data types (ADTs), are the most promising programming-language idea to support software engineering. The ADT is the basis for the "information-hiding" design philosophy [17] that makes software easier to analyze and understand, and that can hope to support maintenance and reuse. There are several formal specification techniques for ADTs [12], a growing number of language implementations of the idea [10], and accepted theories of ADT correctness [5, 9]. The ADT is a good setting for work on unit testing and testability [14].

However, for all the ADT's promise, fundamental problems remain concerning ADTs, their specifications, and implementations. In this paper we address the problem of checking agreement between an ADT's formal specification and its implementation. We show how to write C++ classes and their formal specifications so that the implementation is automatically checked against the specification during execution. Thus the specification serves as an "effective oracle," and in any application the self-checking ADT cannot deviate from specified behavior without the failure being detected.

Because the specification oracle may be inefficient in comparison to the C++ implementation, its use may be confined to prototyping and the testing phase of software development. However, for some applications the specification and implementation may be viewed as independent "versions" of the same software, which continually check each other. Since both were produced by human beings, both are subject to error, but their utterly different form and content suggest that there is minimum chance of a common-mode failure [16].

The central idea that allows self checking is to implement, as part of the C++ code, the mapping between concrete implementation states and abstract specification objects.

2 Self-checking ADTs

We describe ADTs that check their implementations against specifications at run time, and give a simple illustrative example.

2.1 Automatic Testing

The C++ implementation side of our system is available "off the shelf." The second component we require
is a formal axiomatic specification. The specification form determines our ability to mechanically check for properties like the consistency of a newly added operation, and plays an essential role in efficiently checking for abstract equality when the specification serves as an implementation oracle. We use a rewrite system restricted so that the desirable properties of confluence and termination can be obtained from the syntactic forms [2].

The user of our system must supply one additional component, of central importance in our scheme: a “representation” mapping between the concrete data structures of C++ instance variables, and the abstractions of the specification. It is a major weakness of present ADT theory that the representation mapping is nowhere explicit. The existing theory is framed so that the implementation is correct if there exists an appropriate representation [7]. But in practice, the implementor must have a representation in mind. That there is no way to formally record and maintain this early, crucial design decision is a flaw in all existing ADT design methodologies.

Having written an axiomatic specification, a C++ class, and an explicit representation mapping, the user may now test the composite ADT using any unit-test technique. For example, a conventional driver and code coverage tool could be used to ensure that the C++ code has been adequately tested according to (say) a dataflow criterion [18]. Whatever techniques are used, the system will serve as the test oracle that all existing testing systems lack. It determines the correctness of each operation invoked, according to the specification. Alternately, the user might decide to test the ADT “in place,” by writing application code using it, and conducting a system test on the application program with embedded ADT. During this test, the ADT cannot fail without detection.

2.2 Example: A Small Integer Set

To illustrate our ideas, we use the class “small set of integer” (first used by Hoare [13] to discuss the theory of data abstraction). The signature for this ADT is shown in figure 1.

2.2.1 The Specification

Our specification notation is similar to those of several other “algebraic” specifications, such as ACT ONE [5] or Larch [11]. User-defined sorts are specified by an enumeration of constructors each with its arity and parameter types, followed by axioms. An axiom is a rewrite rule of the form

\[ l \rightarrow r \in c, \]

where \( l \) and \( r \) are respectively the left and right sides of the rule and \( c \) is an optional condition [15]—a guard for the application of the rule. The symbol "?" on the right side of a rule denotes an exception.

The ADT \( \text{intset} \) is specified as follows:

\begin{verbatim}
sort intset
constructor
  empty(integer, integer)
    -- maxsize and element upper bound
  insert(integer, intset)
    -- add element to set
\end{verbatim}

\footnote{This name was used by Hoare in his foundational paper [13]. Perhaps “abstraction mapping” is the more common name, which also better expresses the direction.}

\footnote{Its semantics can be formalized within the framework of order sorted algebra [8].}
The axioms for member, cardinality, and maxsize will not be given here; we rely on the reader's intuition of these concepts. The style used in the axioms above handles exceptional cases with a "?" axiom, guarded by a constraint defining the exceptional condition. Thus in the empty axiom the exception occurs only if the parameters are inconsistent. Similarly, the first axiom for insert handles the error case in which an attempt is made to insert an element that violates the upper-bound restriction; and, the third axiom for insert handles an attempt to insert a new element into a set that is already at maximum size. The second and fourth insert axioms establish that the normal form for a nest of insertions is in element order, without duplicates.

Specifications are created incrementally using the "stepwise specification by extension" approach [5]. Each increment adds new operations to a specification. The new specification is a complete and consistent extension of the old one. Two design strategies [2] guarantee completeness and consistency:

The binary choice strategy generates a set of complete and mutually exclusive arguments for an operation. Once we have a left side of a rule we define a set of right sides such that the set of conditions associated with the right sides are mutually exclusive.

The recursive reduction strategy uses a mechanism similar to primitive recursion, but more expressive, for defining the right side of a rule in a way which ensures termination.

Although these strategies make the task of writing a specification somewhat more difficult, the discipline of stepwise specification is a large gain.

2.2.2 Self-checking Implementation

We consider three implementations: a by-hand implementation, a direct implementation, and a self-checking implementation. A by-hand implementation is C++ code written by a programmer to provide the functionality expressed by the specification. This code is naturally structured as a C++ class in which operations are implemented as class member functions. A by-hand implementation of intset appears as the first example in [20, §5.3.2]. The direct implementation [12] is C++ code generated from the specification by representing instances of abstract data types as ground terms, and manipulating them with the rewrite rules. Term rewriting systems of this kind are well known, for example OBJ3 [10]. The self-checking implementation is the union of the by-hand implementation and the direct implementation with some additional C++ code to check their mutual agreement.

The direct implementation provides a C++ mechanism for computing normal-form terms corresponding to any sequence of specification operations. The by-hand implementation provides a similar mechanism for computing a result using any sequence of its member-function calls. These two computations correspond respectively to the upper (abstract) and lower (concrete) mappings in diagrams such as that displayed in figure 2.

![Figure 2: Commuting diagram for the member operation of the ADT set. The abstract world is above the dashed line, the concrete world is below.](image-url)
a commuting diagram is correct according to the abstract specification there. In figure 2, suppose that the boolean result returned by \texttt{member} is \( m(z, S) \) where \( z \) is an integer value and \( S \) is an \texttt{intset} value. (That is, \( m \) is the function computed by \texttt{member}.) Then the diagram commutes iff

\[
\forall z \in S [R(z) \in R(S) \iff R(m(z, S))].
\]  

(2)

Thus to check that the diagram commutes for a particular state value (lower left of figure 2), requires only an implementation of the representation function \( R \). The self-checking implementation comprises the C++ code of both by-hand and direct implementations, plus a C++ representation function, and appropriate calls connecting them. The locus of control lies in the code of both by-hand and direct implementations, plus the self-checking implementation. The additional private entities declared first, in the example of type \texttt{absset}, which is the type mark for a set in the direct implementation. The additional variable \texttt{abstract} contains values of sets from the direct implementation. The additional function \texttt{conc2abstr} is the representation mapping; it takes as input parameters the instance variables of \texttt{intset} and returns the corresponding \texttt{absset} instance.

\begin{verbatim}
// Declaration of the self-checking class.
class intset {
    absset abstract;
    // abstract version of this class
    absset conc2abstr();
    // representation function
    // Below this line the class is identical
    // to Stroustrup, p. 146ff
    int cursize, maxsize;
    int *x;
    public:
        intset(int m, int n);
        // at most m ints in 1..n
        ...  // (most of the code omitted)

    Member functions of the self-checking implementation differ from the corresponding ones in the by-hand implementation only by the addition of two statements just before each function return. For example, the self-checking member function implementing the specification operation \texttt{empty} follows:

    intset::intset(int m, int n)
        // at most m ints in 1..n, Stroustrup, p. 147
        {
            if (m<1 || n>m) error("illegal intset size");
            cursize = 0;
            maxsize = m;
            x = new int[maxsize];
            // Additional statements for self-checking:
            // compute abstract value and self-check
            abstract = empty(m,n);
            selfcheck;
        }
    
    The direct-implementation function \texttt{empty} is called and its result—a normal form encoded in the data structures of the direct implementation—saved in the added variable \texttt{abstract}.

    The macro \texttt{selfcheck} compares the value stored in \texttt{abstract} and that computed by the representation function \texttt{conc2abstr}.

    The C++ version of the representation function is straightforward, starting with an \texttt{empty} abstract set and adding elements from the concrete version one at a time to calculate the corresponding abstract set.

    absset intset::conc2abstr()
    {
        absset h = empty(maxsize,MAXINT);
        for (int i = 0; i < cursize; i++)
            h = ::insert(x[i],h);
        return h;
    }

2.2.3 Errors in the Small Integer Set ADT

When we tested a self-checking implementation corresponding to Stroustrup’s by-hand implementation [20, §5.3.2] (augmented by the member functions needed to implement auxiliary functions like cardinality), we uncovered two defects in the class and its documentation. First, although one is led to believe that a maximum is imposed on the magnitudes of elements that can be inserted in a set, in fact there is none. Second, although it is not documented, the
\end{verbatim}
set is stored with duplicates, which is likely to cause trouble for anyone attempting to reuse and extend the class. These difficulties, discovered in the first example of a well known textbook, suggest that self-checking implementations will be useful in exposing defects.

3 An Automatic Testing System

The direct implementation is a straightforward C++ implementation of term rewriting, which can be automatically constructed from any specification. Our restricted form of axioms guarantees that the resulting program will always terminate when asked to compute a normal form, and that the normal forms will be unique.

The additions to the by-hand implementation are not always quite as simple as those in the example, but they also can be automatically generated.

Thus from a given specification and by-hand implementation, it is possible to automatically generate a self-checking implementation, as indicated in figure 3 for the example of section 2.2.

4 Relation to Previous Work

Although we draw on work in executable specifications using rewrite rules [15] and automatic programming [3], our ideas are most similar to other testing systems. GYPSY [1] uses run-time assertion checking, but the assertions are developed largely by hand and the system predates ADTs. DAISTS [6] is the direct inspiration for our work; it compares implementation and specification in the concrete domain, while we make the comparison in the abstract domain.

Frankl and Doong [4] describe a system that uses rewriting to obtain one (abstract) test case from another, so that the results of an implementation can be compared on these cases. Sankar [19] uses a theorem prover to attempt to prove abstract equality between all terms an implementation generates. Both of these systems seem less straightforward than ours, because they lack the explicit representation function and the specification restrictions to guarantee rewriting termination.

5 Discussion

Ours is not a proof system that verifies an ADT implementation, nor an automatic programming system that attempts to generate code from a specification. But it can be implemented efficiently with current technology, as those cannot. Practical implementation tricks like updating in place, which are the bane of those systems, cause no difficulty whatsoever. Our system will not allow the combination of specification, implementation, and representation function to be inconsistent in the sense of a (non)commuting diagram, without detection.

It solves the "oracle problem" uniformly acknowledged in the testing literature. An effective oracle makes random testing without human intervention a real possibility.

It allows an algebraic specification to be used in multiversion programming as a program with extreme design diversity (compared to a conventional implementation).
However, our scheme is not without its costs: (1) A precise algebraic specification must be written by hand. (2) A representation function must be explicitly coded. We believe that the restrictions placed on our specification form are worth the trouble they cause, because without them we cannot depend on the direct implementation as oracle. The difficulty of observing restrictions is certainly small compared to the difficulty of writing any precise specification. The explicit representation function we view as a blessing in disguise. Unless the programmer has a detailed and accurate idea of this function, it is impossible to write correct member functions that implement the specification's operations. What better way to force this understanding than to insist that it be put into code? What better way to protect against changes that are inconsistent with the representation than to make use of its code? (These issues arose in testing the example in section 2.2.2.)

References


