Testing Ada Abstract Data Types using Formal Specifications

Stéphane Barbey and Didier Buchs

Swiss Federal Institute of Technology in Lausanne
Software Engineering Laboratory
EPFL–DI–LGL
CH–1015 Lausanne
Switzerland

e-mail*: {Stephane.Barbey, Didier.Buchs} @ di.epfl.ch*

Abstract. The purpose of this paper is to present an experiment with formal methods for the black-box testing of reusable abstract data types (ADTs). We propose to test a family of software components by re-engineering a formal specification from an existing Ada implementation, using algebraic specifications. From this well-defined basis, we generate test sets automatically, by constructing an exhaustive set of formulae that prove the property preservations of a program with respect to its specifications, and by reducing this set to a finite size by applying reduction hypotheses. Once the selection of a finite test set has been performed, we show how to build the oracle, the decision procedure for the success or the failure of a test set. Finally, we discuss the advantages of test sets generated from the formal specification over those defined by a programmer, based on his intuitive understanding of the behavior of the ADT.

Keywords. Ada, ADT, formal specifications, reengineering, testing, automatic test sets generation.

1 Introduction

Reusable components need testing more than any other component. They are used and reused in various contexts, sometimes differing significantly from those that the component's developer had in mind. There is therefore a need for robust and well-tested components, and for rigorous testing methods.

The purpose of this paper is to introduce formal methods for the functional (i.e. black-box or specification-based) testing of reusable abstract data types (ADTs). We believe that formal methods contribute to making testing more efficient. The construction and execution of tests is expensive, both in terms of time (delayed program release) and costs (human resources,...). Thus the replacement of manual testing by automated methods is a laudable goal. It is possible to automate several of the different parts of functional testing (automatic generation of test sets, automatic execution, automatic interpretation). Some of these automations have been surveyed in [12]. In this paper, we will focus on the generation of test sets from algebraic specifications. Since, for specification-based testing, the test sets do not depend on the choice of the implementation, they can be automatically generated from the specifications.

Our testing framework, inspired from the work of Gaudel, Bernot and Marre [3], presents the principles that must be applied to automatically select valid tests, how to apply such tests to programs, as well as how to decide if the tests succeed. It is clear that these test sets must be finite, and consequently it is not possible to completely prove the adequacy of the programs with respect to their specifications unless some hypotheses are applied to bound the number of generated tests. These hypotheses are defined and justified in the theory of Bernot [2].

We also consider the oracle problem in the process of automatically generating test sets. An oracle is the decision process for the success or the failure of a test set, i.e. the decision whether or not a program satisfies its specification. However, due to the abstract nature of the components, it is impossible to observe their content without intrusion into the component [9]. The oracle problem is therefore to define whether or not a test returns a correct result. We consider how to limit our observation to primitive sorts (for instance, the boolean type, the integer types, or even the private types) rather than direct comparison between values of the tested abstract data type.

As an experiment, we generate a test sets for a family of associative tables. First, we reengineer the family of components by writing a formal specification using algebraic specifications. From this well-defined basis, we generate test sets automatically. Finally, we briefly discuss the advantages of test sets generated on the basis of the formal specification over those defined by a programmer interpreting the intended behavior of the ADT.

2 The Tested Abstract Data Types

The Software Engineering Laboratory at EPFL has developed and is continually extending a library of components for Ada, including abstract data types (ADTs), text processing utilities, grammar parser generators, and system interfaces [13]. For instance, most kinds of container data structures (stacks, tables, sets,...) have been defined, and implemented as generic packages. However, the behavior of these components was not formally defined by algebraic specifications. The components are used for teaching [14] and in industry. Most of them have been released as publicly available software.

For the container ADTs, implementation considerations [8] have led to propose different variants (a family) of the same component (i.e. having the same functional behavior) in order to solve the problems of parameter passing, etc. The number of variants depends on the possible time and space behavior (the storage properties,...) and the class of types (private, discrete, limited,...) of the contained objects.

The design of these ADTs is based on the “shopping list” approach [11]. The number of operations provided by the component exceeds the number of atomic operations.

Some unessential features have been added for the sake of easing the work of the user by providing the most useful combinations of atomic operations.

2.1 An Interface to a Table ADT

We have chosen to perform our experiment on a family of associative tables. These packages provides associative tables of unlimited dynamic size with entries of type (KEY_TYPE, VALUE_TYPE), where KEY_TYPE and VALUE_TYPE are specified by generic parameters. Such a couple is also called an item. For KEY_TYPE, the relational operations LESS and EQUALS are also specified by generic parameters. These operations are expected to fulfill the canonical consistency conditions. A table may not contain duplicate items, that is having the same key. Two items (k1, v1) and (k2, v2) have the same key if and only if EQUALS (k1, k2) returns true. There are no generic subprograms associated with VALUE_TYPE.

Four variants of tables have been implemented to consider the possible classes of types of KEY_TYPE and VALUE_TYPE, which can either be static or dynamic. In our terminology, a static type is a type which is neither a limited type nor an access type. When an actual generic access type is associated with a generic static type, objects are shared, i.e. only the access value is be stored, without copying the accessed object. On the opposite, a dynamic type may be limited or an access type, and we assume that every object of a dynamic type has an implied initial value. Therefore, depending on the nature of the types KEY_TYPE and VALUE_TYPE, one of the provided packages ha s to be instantiated (TABLE_OF_DYNAMIC_KEYS_AND_DYNAMIC_VALUES_G, TABL E_OF_STATIC_KEYS_AND_DYNAMIC_VALUES_G, TABLE_OF_DYNAMIC_KEYS_AND_ STATIC_VALUES_G, TABLE_OF_STATIC_KEYS_AND_STATIC_VALUES_G). However, the number of variants has no influence on the generation of test sets. Since the generated test sets do not depend on the choice of the implementation, we generate one single test set for this family.

For instance, here is an excerpt of the package specification of a table of static keys and values.

```
generic
   type KEY_TYPE is private;
   with function LESS (LEFT, RIGHT: KEY_TYPE) return BOOLEAN;
   -- Defines ordering of keys.
   with function EQUALS (LEFT, RIGHT: KEY_TYPE) return BOOLEAN;
   -- Defines equality of keys.

   type VALUE_TYPE is private;
package TABLE_OF_STATIC_KEYS_AND_STATIC_VALUES_G is

   type TABLE_TYPE is limited private;

   DUPLICATE_ITEM_ERROR,
   MISSING_ITEM_ERROR: exception;

   procedure ASSIGN (DESTINATION: in out TABLE_TYPE; SOURCE: in TABLE_TYPE);
   --+ Begins by a call to DESTROY (DESTINATION) and then copies SOURCE into
   --+ DESTINATION.

   procedure INSERT(TABLE: in out TABLE_TYPE;
                    KEY: in KEY_TYPE;
```

```
                    VALUE: in VALUE_TYPE);
    --+ Inserts the couple (KEY, VALUE) into TABLE.
    --+ If an entry with the given key is already in the table, then exception
    --+ DUPLICATE_ITEM_ERROR is raised.

    procedure REPLACE_VALUE(TABLE: in out TABLE_TYPE;
                            KEY: in KEY_TYPE;
                            VALUE: in VALUE_TYPE);
    --+ An entry having key KEY is searched for in TABLE. The given VALUE then
    --+ replaces the previous one.
    --+ If there is no entry with the given key, the exception
    --+ MISSING_ITEM_ERROR is raised.

    procedure REMOVE (TABLE: in out TABLE_TYPE;
                    KEY: in KEY_TYPE);
    --+ Removes the entry with key KEY from TABLE
    --+ If there is no entry with the given key, the exception
    --+ MISSING_ITEM_ERROR is raised.

    function SIZE (TABLE: in TABLE_TYPE) return NATURAL;
    --+ Returns the number of entries currently in TABLE.

    function IS_EMPTY (TABLE: in TABLE_TYPE) return BOOLEAN;
    --+ Returns TRUE if and only if the TABLE is empty.

    function IS_PRESENT (TABLE: in TABLE_TYPE; KEY: in KEY_TYPE)
        return BOOLEAN;
    --+ Returns TRUE if and only if an ITEM with key KEY is in TABLE.

    function VALUE (TABLE: in TABLE_TYPE; KEY: in KEY_TYPE) return VALUE_TYPE;

    generic
        with procedure ACTION(KEY: in KEY_TYPE;
                              VALUE: in VALUE_TYPE) is <>;
    procedure TRAVERSE_ASC_G (TABLE: in TABLE_TYPE);
    --+ The entries in TABLE are visited in ascending order of their key
    --+ values. Procedure ACTION is applied on each entry within TABLE.
    --+ For your actual procedure ACTION, you must not use a procedure
    --+ which modifies the traversed table.

private
    --+ ...
end TABLE_OF_STATIC_KEYS_AND_STATIC_VALUES_G;
```

Fig. 1. An excerpt of an Ada package specification of an associative table

3 Formal Specifications

Both the theory of abstract data types and the theory of algebraic specifications find their origin in the impossibility to manage and to develop large software systems without methods and tools. They are both aimed at reducing the complexity of developing software by providing, for ADTs, a clearly designated and safe development unit, and, for formal specifications, a means for concise and rigorous behavior descriptions, and thus testable specifications.

3.1 Reengineering the ADT into a Formal Specification

Since the specification of the table ADT were written in natural language, we have engaged our experiment by reengineering the Ada specifications into a formal specification using the algebraic part of the CO-OPN specification language [4][5]. Unfortunately, the algebraic specification could not be translated directly from the Ada specification; since CO-OPN is mainly based on the functional paradigm, it does not provide for all Ada constructs:

- procedures
  All procedures must be translated into functions. The **in out** parameters must be passed as a parameter of the function and be returned by the result of the function.The **out** parameters are the results of the functions.
- generics
  The generic formal parameters must be extracted into a separate signature, to which the actual parameters have to conform. It must be noticed that more constraining parameters can be described in algebraic specifications languages. For instance, we introduce axioms for ensuring the consistency conditions described in 2.1 (the satisfaction of the order relation by the function LESS) (see figure 3).
- exceptions
  Since, like most algebraic specification languages, CO-OPN does not handle exceptions, the conditions that can raise an exception are implemented like functions that return boolean results telling whether the original function would raise the exception or not.

In an effort of simplification, the shopping list approach led us not to specify every operation, but just the atomic operations. Therefore, the Ada operations that consisted in the combination of two or more atomic operations do not appear in the algebraic specification.

3.2 Data Types and Operations

Before going deeper into details we need to introduce some terminology. An abstract data type is the representation of a data structure coupled with a set of operations which can be applied to this data structure (data type) independently of any particular implementation (abstract).

The algebraic approach to specification considers a system in terms of abstract data types represented in the mathematical discourse by a many-sorted algebra together with axioms or equations. An algebra is simply a domain provided with a set of operations. An algebraic specification consists of a quadruplet Spec = <S, F, X, AX> where:

- S is a set of sort names;
- F is a set of operations applicable to the sorts, characterized by their names and their signatures;
- AX is a set conditional axioms, which define the behavior of the sorts; every axiom is composed of an equation possibly preceded by a set of conditions;
- and X is a set of variables, which may appear in the axioms.

The signature of a formal specification is composed of the sets S and F. The signature of a formal specification is similar in content to the information found in an Ada specification, where a type is called a *sort* and a subprogram an *operation*. The *axioms* and the *variables* do not appear in Ada specifications, because Ada specifications do not carry information regarding the behavior of the component.

Here is an excerpt of the signature of the formal specification of an ADT of a table in CO-OPN. The parameter specification Values_Spec defines the signature for the generic type parameter Value_Type (and the implicitly associated equality operator). The generic specification Tables_Spec describes the sort Table_Type based on its parameters (Values_Spec and Keys_Spec), and the imported sort Boolean. The iterator Traverse_Asc_Spec is described in a similar fashion using the generic specification Traverse_Asc_Spec and the parameter specification Actions_Spec.

```
par spec Values_Spec;
use
   Boolean_Spec;
sorts
   Value;
operations
   Equal: Value, Value -> Boolean;
axioms
   ...
end; % Values_Spec %

generic spec Tables_Spec (Keys_Spec, Values_Spec);
use
   Keys_Spec, Values_Spec, Boolean_Spec;
sorts
   Table;
generated by
   Empty: -> Table;
   Insert: Table, Key, Value -> Table;

operations
   Assign: Table -> Table;
   Duplicate_Item_Error: Table, Key -> Boolean;

   Replace_Value: Table, Key, Value -> Table;
   Missing_Item_Error: Table, Key -> Boolean;

   Remove: Table, Key -> Table;

   Size: Table -> Natural;
   Is_Empty: Table -> Boolean;
   Is_Present: Table, Key -> Boolean;
   Value: Table, Key -> Value;
axioms
   ...
end; % Tables_Spec %

par spec Actions_Spec;
use
   Keys_Spec, Values_Spec, Natural_Spec, Boolean_Spec;
operation
   Action: Key, Value, Natural -> Boolean;
end; % Actions_Spec %

generic spec Traverse_Asc_Spec (Tables_Spec, Actions_Spec)
use
   Tables_Spec, Actions_Spec, Keys_Spec;
operation
   Traverse_Asc: Table -> Table;
end; % Traverse_Asc_Spec %
```

Fig. 2. An excerpt of the signature of the formal specification of an associative table

3.3 The Properties of an ADT

The semantics of an algebraic specification consist of the class of possible programs that satisfy all the axioms of the specification. A program satisfies an axiom if it always satisfies the equation when the conditions are satisfied. Therefore, testing a program against its specification is to check that the program is included in the semantics of the specification. This definition of satisfaction implies that every axiom of the specification can be checked in isolation.

The properties of a specification are equations over the terms of the specifications. They are given by a set of deductive theorems DTh (Spec) [7]. However, this set is not limited to the theorems deduced from the user-defined conditional axioms; some properties are also deduced from the logical axioms and the equality axioms.

We give below the axioms of the Values_Spec and Tables_Spec specification.

```
par spec Values_Spec;
   ... % see above%
axioms
   ax_Equal_0: Equal (X, X) = true; % reflexivity %
   ax_Equal_1: Equal (X, Y) = Equal (Y, X); % symmetry %
   ax_Equal_2: Equal (X, Y) = B & Equal (Y, Z) = C
      => Equal (X, Z) = and (B, C); % transitivity %
where
   X, Y, Z: Value;
   B, C: Boolean;
end; % Values_Spec %

generic spec Tables_Spec(Keys_Spec, Values_Spec)
   ... % see above %
axiom
   ax_Assign_0: Assign(T) = T;

   ax_Duplicate_Item_Error_0: Duplicate_Item_Error(Empty,K) = false;
   ax_Duplicate_Item_Error_1: Equal(K1,K2) = true
      => Duplicate_Item_Error(Insert(T,K1,V),K2) = true;
   ax_Duplicate_Item_Error_2: Equal(K1,K2) = false
      => Duplicate_Item_Error(Insert(T,K1,V),K2)=Duplicate_Item_Error(T,K2);

   ax_Replace_Value_0: Replace_Value(T,K,V) = Insert(T,K,V);

   ax_Remove_0: Remove(Empty,K) = Empty;
   ax_Remove_1: Equal(K1,K2) = true
      => Remove(Insert(T,K1,V),K2) = Remove(T,K2);
   ax_Remove_2: Equal(K1,K2) = false
      => Remove(Insert(T,K1,V),K2) = Insert(Remove(T,K2),K1,V);


   ax_Size_0: Size(Empty) = 0;
   ax_Size_1: Missing_Item_Error(T,K) = true
      => Size(Insert(T,K,V)) = succ(Size(T));
   ax_Size_2: Missing_Item_Error(T,K) = false
      => Size(Insert(T,K,V)) = Size(T);

   ax_Is_Empty_0: Is_Empty(Empty) = true;
   ax_Is_Empty_1: Is_Empty(Insert(T,K,V)) = false;

   ax_Is_Present_0: Is_Present(T,K) = Duplicate_Item_Error(T,K);

   ax_Value_1: Equal(K1,K2) = true => Value(Insert(T,K1,V),K2) = V;
   ax_Value_2: Equal(K1,K2) = false =>
      Value(Insert(T,K1,V),K2) = Value(T,K2);
```

```
where
   T: Table;
   K,K1,K2: Key;
   V,V1,V2: Value;
end; % Tables_Spec %
```

Fig. 3. An excerpt of the axioms of an associative table

4 The Theory of Test Selection

The theory of test selection consists in selecting from a possibly infinite set of formulae, corresponding to all the properties of a program, a finite set of ground formulae (equalities without variables) which is sufficient to prove the property preservations of a program with respect to its specification. This infinite set of formulae is called the exhaustive test set. The reduction of the exhaustive test set to the ground formulae is performed by applying hypotheses to the set. Those hypotheses, called reduction hypotheses, define selection strategies and reflect common test practices [3]. The goal of the strategies is select formulae showing counter-examples to the program validity, i.e. to exhibit the presence of errors.

Once a finite set of formulae is selected, it is necessary to have a decision procedure for the success or the failure of the test set. This process is called an oracle [15]. The oracle decision can be an undecidable problem depending on the complexity of the proof of the equality [2] between expressions in programming languages. (For example, there may be no way to express the equality between two objects of a limited type.)

Thus, we can define the selection of a test set as the iterative refinement of testing context defined as a triplet (H,T,O) where T is the test set, H is the set of hypotheses and O is the oracle. The introduction of a new hypothesis h provides a new testing context (H',T',O'), where $H' = H \cup h$, $T' \subseteq T$ and O' is more defined than O. This refinement of the context (H,T,O) in (H',T',O') induces a pre-order between contexts ((H',T',O') $\leq$ (H,T,O)). The starting point of the context refinement is the testing context ($\varnothing$, T, $\perp$), where T is the exhaustive test set, no hypotheses are defined, and the oracle is undefined. The context refinement is performed iteratively until we reach a test set that is finite, decidable, and preserves the properties of the exhaustive test set.

5 An Example of Test Set Generation

We show through an example how to produce a finite test set for an axiom taken from the algebraic specification of the associative tables:

```
ax_Remove_1: Equal(K1,K2) = True
   => Remove(Insert(T,K1,V),K2) = Remove(T,K2);
```

Fig. 4. Axiom under testing

Of course those axioms cannot be tested unless the generic specifications are instantiated. In the following discussion, we use Natural_Spec for both the parameter Values_Spec and the Keys_Spec.

5.1 The Exhaustive Test Set

All the properties induced by the axiom could be generated by replacing the variables by all possible terms. These properties are included in DTh (Spec) and must be valid into the program to be tested. For the axiom ax_Remove_1, the exhaustive test has the following structure:

```
test 1:
    Remove(Insert(Empty,zero,zero),zero) = Remove(Empty, zero);
test 2:
    Remove(Insert(Empty,succ(zero),zero),succ(zero)) =
        Remove(Empty, succ(zero));
test 3:
    Remove(Insert(Empty,succ(succ(zero)),zero),succ(succ(zero))) =
        Remove(Empty, succ(succ(zero)));
...
test n:
    Remove(Insert(Empty,zero,zero),zero) = Remove(Empty, zero);
test n+1:
    Remove(Insert(Empty,zero,succ(zero)),zero) = Remove(Empty, zero);
test n+2:
    Remove(Insert(Empty,zero,succ(succ(zero))),zero) = Remove(Empty, zero);
...
test p:
    Remove(Insert(Insert(Empty,zero,zero),zero,zero),zero) =
        Remove((Insert(Empty,zero,zer, zero);
test p+1:
    Remove(Insert(Insert(Insert(Empty,zero,zero),zero,zero),zero,zero),zero)=
        Remove(Insert(Insert(Empty,zero,zero),zero,zero), zero);
...
```

Fig. 5. The test set for the axiom ax_Remove_1

However, the exhaustive test set is not finite and useless as in the example. We must introduce reduction hypotheses to refine it.

5.2 The Chosen Reduction Hypotheses

This reduction is achieved by applying hypotheses preserving the validity of the test set. The two possibly suitable hypotheses are the following:

- Regularity hypothesis
  If a program behaves correctly for an axiom A instantiated to a set of terms lower in complexity than a defined bound, then it should behave correctly for all possible instantiations.

  This hypothesis is used to instantiate the variables of the tested sort. It helps to limit the size of the test set by making assumptions on one term of a certain complexity may represent all terms of the same complexity or lower in complexity.
- Uniformity hypothesis
  If a program behaves correctly for an axiom A instantiated to a particular term T of a sort s $\in$ S, then every instantiation of the axiom A with terms of the sort s behaves correctly. As a consequence, if the axiom A is correct for one specific instantiation, it should be correct for all possible instantiations.

  This hypothesis corresponds to random testing, and is used to instantiate the variables of the primitive sorts (i.e. a sort used by the tested sort).

In the case of the table ADT, we apply the uniformity hypothesis on the keys and values and the regularity hypothesis on the tables. The regularity hypothesis produces a test set in which all tables are of a given size, lower than a predefined constant, including the cases of the empty table and the table of one item of the manually generated test set. Some constraints, expressed by equalities, remain after applying the regularity hypothesis. These constraints can be solved by applying the uniformity hypothesis which provides for the random choice of items into the data structure. In our example, the variables of the sorts Keys and Values are instantiated by applying the uniformity hypothesis.

For instance, by applying the regularity hypothesis, the test set for the axiom ax_Remove_1 can be reduced to 12 tests, some of the constraints of which are not solved. For example, in the following solution, no values have been given to _v0, _v1 and _v3, because the uniformity hypothesis has not been applied.

```
FINAL BINDING:
K1:key = k(0)
K2:key = k(0)
T:table = insert(_v0:table,k(s(_v1:nat)),_v2:nat)

REMAINING CONSTRAINTS = {
 remove(insert(_v0:table,k(s(_v1:nat)),_v2:nat),k(0)) =
   insert(_v3:table,k(s(_v1:nat)),_v2:nat),
 remove(_v0:table,k(0)) = _v3:table
}
```

Fig. 6. Example of remaining constraint

By applying the uniformity hypothesis and the regularity hypothesis, all the constraints can be solved and the test set can be reduced to 6 tests.

```
FINAL BINDING:
K1:key = k(0)
K2:key = k(0)
T:table = empty
V:nat = 0

FINAL BINDING:
K1:key = k(sucf(1))
K2:key = k(sucf(1))
T:table = insert(insert(empty,k(0),sucf(1)),k(sucf(1)),sucf(1))
V:nat = 0

FINAL BINDING:
K1:key = k(sucf(2))
K2:key = k(sucf(2))
T:table = insert(insert(insert(insert(insert(insert(empty,k(sucf(1)),sucf(3)),
   k(sucf(2)),sucf(5)),k(0),0),k(0),sucf(1)),k(0),sucf(1)),k(0),0)
V:nat = sucf(4)

FINAL BINDING:
K1:key = k(sucf(3))
K2:key = k(sucf(3))
T:table = insert(insert(insert(empty,k(sucf(1)),0),k(0),sucf(3)),k(sucf(1)),0)
V:nat = 0

FINAL BINDING:
K1:key = k(0)
K2:key = k(0)
T:table = insert(insert(empty,k(sucf(1)),0),k(sucf(2)),0)
V:nat = 0
```

```
FINAL BINDING:
K1:key = k(sucf(1))
K2:key = k(sucf(1))
T:table = insert(empty,k(sucf(2)),0)
V:nat = sucf(1)
```

Fig. 7. Generated test set for the axiom ax_Remove_1

Since some of the choices are made at random, two generations of the same query are likely to instantiate the variables differently.

The final test set is obtained by applying the different bindings on figure 7 to the axiom on figure 4.

5.3 The Operational Method

In practice, the principles previously presented are applied with the help of an equational logic programming environment, LOFT. Its purpose is to solve queries expressed in equational logic and composed of the conjunction of the equations that must be satisfied.

For example, the test set on figure 7 is the result of the following query:

```
equal(K1, K2) = true, remove(insert(T, K1, V), K2) = remove(T, K2),
?(is_a_table(T)), ?(is_a_key(K1))
```

Fig. 8. A query for the generation of the test set for ax_Remove_1

The two first equations are directly taken from the axiom. The two last equations express uniformity hypotheses.

The environment is made of a resolution mechanism which tries to solve the query by substituting terms to the variables of the query. Thus, the query is extended with equations until the system can find ground terms as solutions (i.e. there are no more variables to substitute, but only generators). Extending the query corresponds to adding hypothesis. LOFT allows to introduce several ways to control the resolution mechanism, for instance:

- narrowing (a variant of the resolution principle for equational logic),
- 'wait' mechanism (delaying evaluation of clauses in the test set selection process),
- random choice strategy for the choice of the axiom to be applied,
- simplification mechanism using rewriting,
- bounded resolution.

Those mechanisms have been proposed by Marre [10] and implemented in the tool LOFT.

5.4 Writing the Oracle

Given the correspondence of the Ada constructs and the algebraic specification language, we are able to build an oracle. For instance, in the case of exceptions, both sides of the axiom equality must be encapsulated in order to capture the exception and translate it in a boolean value used to establish the validity of the axiom (the axiom is defined using a boolean value instead of an exception). For Ada functions or procedures, the oracle is a direct translation of the axioms into Ada conditional statements.

The problem of the oracle is solved by building the equality of non-observable sorts on the equality defined on the primitive sorts, rather than by direct comparison between values of the tested abstract data type, which may be non available. For an equality between two non-observable values, a composition of operations the result of which is observable (an observable context), is added "on top of" each member of the equality. Therefore, the observation of a non-observable equality is performed through a set of observable equalities corresponding to the addition of observable contexts "on top of" the right and the left member of the non-observable equality.

6 Advantages of the Automatic Test Sets Generation

When the example component was developed, a test program was written, consisting of more than 3200 lines of Ada code. It covers all primitives in each particular critical situation, such as an empty structure, a table of one item, of a given size (>1), etc. However, this test program is based on an intuitive understanding of the specification, and does not ensure that testing is complete. Furthermore, the operations are treated in blocks, and it is sometimes difficult to understand which are the subprograms tested, and which are utilities.

The automatically generated test set does not have all these drawbacks. It is based on rigorous semantics, which compels wrong interpretations of the behavior of the component. It systematically tests all the axioms (properties) of the operations acting on the tables, providing a complete testing of the specification. For each operation, a complete and independent test set is generated, avoiding misleading combinations of operations.

However, according to Spence and Meudec [12], this approach may have distinct problems: the difficulty of writing axiomatic specifications for complex data types, and the large number of tests generated. Practical experiences at an industrial level, for example the application to an automatic subway [6], have shown that the formal specifications may be used with success for complex problems. The second problem may be solved by automating the process even further, for example by an automatic generation of the oracle.

7 Conclusion

In this paper we have introduced formal specifications, and we have shown how to reengineer an Ada component into an algebraic specification. Then, we have described the automatic generation of test sets from a formal specification. We have described the selection theory, and we have shown through a concrete example an usage of this theory. We also have briefly discussed the advantages of the automatic test set generation over manually written test sets.

We believe that using formal methods for generating test sets results in more efficient test sets, due to the systematization of the generation.

In the future, we will study the possible extensions of the proposed test method to object-oriented specifications and programs [1] and in particular to Ada 9X, by generating test sets for hierarchies of types (tagged types, class-wide types) and for the concurrent aspects of Ada 9X (task types and protected types).

8 Acknowledgments

The authors would like to thank Bruno Marre for his help with the tool LOFT and the theory behind it. The work reported in this paper is supported by the Swiss National Science Foundation (Nationalfonds), grant number 21-36038.92.

9 References

[1] Stéphane Barbey and Alfred Strohmeier. The problematics of testing object-oriented software. In M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM '94 Second Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK, July 26-28 1994.

[2] Gilles Bernot. Testing against formal specifications: A theoritical view. In *TAPSOFT '91*, number 494 in Lecture Notes in Computer Sciences. Springer Verlag, April 1991.

[3] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6(6):387–405, November 1991.

[4] Didier Buchs, Jacques Flumet, and Pascal Racloz. SANDS: Structured algebraic net development system. In Buy Ugo, editor, *14th International Conference on Application and Theory of Petri Nets, Tool presentation abstracts*, pages 25–29. Chicago, USA, June 1993.

[5] Didier Buchs and Nicolas Guelfi. A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aahrus, Denmark, June 1991.

[6] Pierre Dauchy, Marie-Claude Gaudel, and Bruno Marre. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244, June 1993. North Holland, Elsevier Science Publishing Company.

[7] Hartmut Ehrig and Bernd Mahr. *Fundamentals of algebraic specification 1: equations and initial semantics*. EATCS Monographs on Theoritical Computer Science. Springer-Verlag, March 1985.

[8] Claude Genillard, Norbert Ebel, and Alfred Strohmeier. Rationale for the design of reusable abstract data types implented in Ada. *ACM Ada Letters*, 9(2):62–71, March 1989.

[9] J. Liddiard. Achieving testability when using Ada packaging and data hiding methods. *Ada User*, 14(1):27–32, March 1993.

[10] Bruno Marre. *Sélection automatique de jeux de tests à partir de spécifications algébriques en utilisant la programmation logique*. PhD thesis, LRI, Université de Paris XI, Orsay, France, January 1991.

[11] Bertrand Meyer. *Reusable Software: the base object-oriented component library*. The Object-Oriented Series. Prentice-Hall, 1994.

[12] I. Spence and C. Meudec. Generation of software tests from specifications. In M. Ross, C.A. Brebbia, G. Staples, and J. Stapleton, editors, *SQM '94 Second Conference on Software Quality Management*, volume 2, pages 517–530, Edinburgh, Scotland, UK, July 26-28 1994.

[13] Alfred Strohmeier, editor. *Ada Software Components*. EPFL, 2nd edition, 1992.

[14] Alfred Strohmeier. Use of a software component library in student projects. In A. Finkelstein and B. Nuseibeh, editors, *ACM/IEEE International Workshop on Software Education (ICSE)*, pages 319–326, Sorrento, Italia, 1994.

[15] Elaine J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.