Component Assessment using Testing

Marisa A. Sánchez, Elsa Estévez, and Pablo Fillottrani

Departamento de Ciencias de la Computación
Universidad Nacional del Sur,
Avda. Alem 1253, 8000 Bahía Blanca,
Telephone +291 459 5135, Fax +291 459 5136,
mas[ece,prf]@cs.uns.edu.ar

Abstract. In the last years software researchers have been looking for ways of assembling systems in a style of software construction similar to "LEGO blocks". Software components are reusable building blocks for constructing software systems. Component-based development may greatly increase the productivity of software engineers and improve the quality of software. There are many issues related with components that are of wide interest both to academics and people from industry. One of them is the retrieval of components that will be assembled into a new system. It is difficult to decide whether an implementation fits on a predefined design. In this work, we consider the problem of assessing that the implementation of a concrete component is consistent with the specification of the desired functionality. We assume that the component source code is not available and we have a formal specification of the system of interest. We propose to perform a dynamic assessment using tests derived from the specification (semantic criteria) but executed using the implementation of the candidate component.

Key words: Software engineering, component retrieval, reusable software, software libraries, testing, algebraic specifications.

1 Introduction

The growing size and scope of systems have changed the traditional practices of software engineering. Building new reliable solutions within a tight schedule is the key point of most current projects. Doug McIlroy predicted that mass-produced components would end the so-called software crisis. He introduced the idea about software reuse, proposing an industry of source code for components "off the shelf". Reuse is a very simple concept: to use the same thing more than once. In software engineering this concept means to use already

developed parts of software to construct new software. Software reuse is the process of building or assembling software systems from predefined components that are designed for reuse. It deals with the production and use of components.

Components and component-based development are approaches that enable practical reuse of software "parts". More and more organizations are turning to components as a way to encapsulate existing functionality, acquire third-party solutions, and build new services to support emerging business processes. The latest technologies for distributed systems support and encourage a component view of application integration and deployment [1]. Furthermore, component-based development provides a design paradigm well suited for todays approach, were the traditional *design and buid* has been replaced by *select and integrate*. All aspects of software design, implementation, deployment, and evolution are affected when a component-based approach is followed. As a result, a software project can be transformed from a development-intensive grind of code writing and bug fixing, to a more controlled assembly process in which new code development is minimized and system upgrade becomes the task of replacement of well-bounded functional units of the system.

For component-based development, a component is much more than a subroutine in a modular programming approach, an object or class in an object-oriented system, or a package in a system model. The notion of component both subsumes and expands on those ideas. A component is used as the basis for design, implementation, and maintenance of component-based systems. Informally, a component may be defined as *an independently deliverable piece of functionality providing access to its service through interfaces* [1]. An *interface* defines a set of properties, methods, and events through which external entities can connect to, and communicate with, the component, [2].

One of the main problems in component based development is the retrieval of the components that will be assembled into the new system. It is difficult to decide whether an implementation fits on a predefined design. Most research in the area of component assesstment is focused on retrieval from a repository based on syntactic characteristics of the components. Significant effort has been placed on the development of interface definition languages (IDL) to support syntactic retrieval criteria. IDLs are necessary, but they only

consider information defined in the interfaces. On the other hand, semantic criteria is fundamental when some components must be composed to perform the desired behavior for the system.

In this work, we consider the problem of assessing that the implementation of a concrete component is consistent with the specification of the desired functionality. There are a number of challenges that arise when attempting to assess the behavior of a component, particularly one that is "off the shelf". First, the source code of the component is generally not available. Second, component behavior is rarely described formally, thus preventing any direct transformation into a model suitable for direct analysis. Additionally, retrieval approaches that address semantic criteria assume that the components in the repository are specified using a formal language [3]. In this paper, we address the problem of assessing a software component when its source code is not available and we have a formal specification of the desired component. Thus, we assume we have a formal specification of the system of interest and a library with components that we would like to retrieve for reuse. We consider RSL (RAISE Specification Language) [4], [5]. However, the approach is valid for any algebraic specification language.

Assume that a specification S_C for the desired component C is created. We propose a systematic approach to generate test cases from the specification where axioms are used as tests. Each axiom of the specification defines some properties of the operations occurring in it. Hence, the axioms are very close to requirements. Also, since the result of the evaluation of a test derived from an axiom is always true (false) there is no problem with constructing and validating the expected results. These tests are executed using the candidate components. Hence, we propose tests very close to what the user really wants (semantic aspects) but executed using the implementation of the component that is been evaluated for reuse.

The paper is organized as follows: Sections 2.1 briefly introduce algebraic specifications. Section 2.2, is devoted to describe our testing approach. We first define how we generate tests and then we discuss the selection of terms and concrete data. We then describe how to perform tests using the implementation of a given component (Section 2.3). Conclusions are presented in Section 3.

2 Description of the Approach

2.1 Algebraic Specifications

An algebraic specification is a pair (Σ, Ax) where Σ is the signature of the specification, and Ax is a set of axioms which define the properties of the operations of Σ. A set of sentences ϕ_Σ is associated with each signature Σ. Among the operations of the signature, there is a subset of generators. Any ground term can be proved equal to a term built from generators only [6]. The other operations are called observer operations.

The semantics of an algebraic specification is defined by a class of $\Sigma-$ interpretations, Int_Σ, and a validation predicate on $Int_\Sigma \times \phi_\Sigma$ denoted by $\models$. For each $\Sigma-$ interpretation A and for each formula ϕ, $A \models \phi$ should be read as A validates ϕ. The class of interpretations validating a specification module SC is called the class of models of SC and is denoted by $Mod(SP)$:

$$Mod(SC) = \{A \in Int_\Sigma \mid A \models Ax\}$$

Let C be an implementation of a component to be tested, which is supposed to implement a specification module $SC = (\Sigma, A)$. Testing C with regard to SC is only possible if the semantics of C and SC are expressible in a common framework. As we are interested in dynamic testing, we have to execute C or a finite subset of its input domain, and interpret outputs with regard to SC. Bernot *et al.* ([7], [8]) developed a theory and a tool to generate practicable test sets based on algebraic specifications. Each axiom of the specification defines some properties of the operations occurring in it. Thus, each ground instance of these axioms is a test of the properties of the operations defined in the specification. Also, since the result of the evaluation of a test derived from an axiom is always true (false) there is no problem with constructing and validating the expected results. The authors introduce the notion that a test data set cannot be considered independently of some hypotheses. Hypotheses are the assumptions that allow us to infer the correctness of a program from the success of the test set and incorrectness from failure. Based on the work of Bernot *et al.* we described a framework for testing implementations of systems specified using an algebraic language RSL [9].

2.2 Generation of Test Cases

RAISE is a product which consists of a method to develop software, a formal specification language, RSL (RAISE Specification Language) and a set of tools. In a RAISE development, we start with an abstract specification, we make some design decisions and produce a new specification which conforms to the previous one. The aim is to construct a more concrete specification and this process is known as a *development step*. So, each development step contributes with new information and more details about the design.

As an example, consider the specification of an abstract data type, a bounded queue. The requirements for a bounded queue are that items may be extracted ("dequeued") only in the order in which they were inserted ("enqueued") and that at any time any number of items up to the maximum may have been enqueued and not yet dequeued. Consider the following Abstract Applicative Specification of a Queue A_QUEUE0:

```
scheme
  A_QUEUE0(P : ELEM_BOUND) =
    hide List_of_Queue, list_of in
      class
        type Queue,  List_of_Queue = {| l: P.Elem* • len l ≤ P.bound |}
        value
          /⋆ generators ⋆/
          empty : Queue,
          enq : P.Elem × Queue ~→ Queue,
          deq : Queue ~→ P.Elem × Queue,
          /⋆ hidden observer ⋆/
          list_of : Queue → List_of_Queue,
          /⋆ observers ⋆/
          is_full : Queue → Bool
          is_full(q) ≡ len list_of(q) = P.bound,
          is_empty : Queue → Bool
          is_empty(q) ≡ list_of(q) = ⟨⟩
        axiom
          [list_of_empty] list_of(empty) ≡ ⟨⟩,
          [list_of_enq]
            ∀ e : P.Elem, q : Queue •
              list_of(enq(e, q)) ≡ list_of(q) ^ ⟨ e ⟩ pre ~ is_full(q),
          [deq_ax]
            ∀ q : Queue •
              deq(q) as (e, q′ )
                post e = hd list_of(q) ∧ list_of(q′ ) = tl list_of(q)
                pre ~ is_empty(q),
```

```
        [enq_defined]
            ∀ e : P.Elem, q : Queue • enq(e, q)
          post true pre ∼ is_full(q)
end
```

Definition of tests As mentioned before, we can produce test cases from an RSL specification. The axioms from the abstract specification give tests. For example, the axioms [list_of_empty], [list_of_enq] and [deq_ax] of A_QUEUE0 give tests. We do not use the axiom [enq_defined] as a test case because given any input test data we might not be able to decide if the test is successful or not because of the lack of an effective oracle for testing termination, which is what **post true** effectively means.

Additionally, we can use definitions of derived functions as if they were written as axioms. The functions $is\_full()$ and $is\_empty()$ are derived, so we use their definitions as axioms giving tests. This allows us to evaluate each operation separately. The test derived from axioms focus on the interactions among functions. We can also write a theory, justify it and use it as another test.

In the following, we describe how to select terms to instantiate the proposed axioms.

Generation of terms Any value of the type of interest can be denoted by a composition of some generators. However, the set of terms of this type is infinite. So, we need to bound the size of the list terms. This can be done if we define a regularity hypothesis. Given a complexity measure of level k on the terms (for instance, the number of operations of the type occurring in a term), we can select only those terms that scores less or equal to k. The regularity hypothesis states that if a formula is valid for all the selected terms, then it is valid for all terms of the algebra. The reader is refered to [8] for a detailed discussion of selection hypothesis.

We have to select terms of the type *Queue*. Given a definition of a complexity measure, we can specify an algorithm to generate the desired terms. Let us use a complexity measure on the terms of this type defined as follows: a term scores the sum of scores of its generators. The scores for the generators are 1 for *empty* and

enq, and 0 for *deq*. We apply a regularity hypothesis (of level 3) and generate the following terms:

1. empty
2. enq(e1,empty)
3. enq(e2,enq(e3,empty))
4. **let** (e,q) = deq(enq(e4,empty)) **in** q **end**
5. **let** (e,q) = deq(enq(e4,empty)) **in** enq(e5,q) **end**
6. **let** (e,q) = deq(enq(e7,enq(e8,empty))) **in**
   let (e′ ,q′) = deq(q) **in** q′ **end end**
7. **let** (e,q) = deq(enq(e9,empty)) **in**
   let (e′ ,q′) = deq(enq(e10,q)) **in** q′ **end end**

Data that does not satisfy the precondition of the axiom is not used. For example, $enq(e, q)$ where q is a full queue. In general, we reject test data when the precondition of the proposed axiom is not provable because it is false or irreducible (as for $deq(empty)$).

Selection of Concrete Data We need to instantiate the terms selected for testing. Data will be needed for any variables universally quantified in the axioms.

Additionally, we should also select data for the imported types in the specification of interest. The variable e included in the axioms is of the imported type $P.Elem$. We can assume that imported types belong from components that have already been assessed. Hence, as proposed in [8], we can use an uniformity hypothesis to select data for the imported types. A uniformity hypothesis states that if a formula is true for some value then it is always true.

Once we have the final specification expressed in RSL and we have generated the test cases we have to derive the test functions from the axioms. For example, we can derive the following imperative version of the test function derived from the axiom [list_of_enq]:

```
list_of_enq : Text × Int* × Int → write any Bool
list_of_enq(code, parameters, e) ≡
    gen_queue(code, parameters) ;
    if is_full() then
        true
    else
        let l = list_of() in enq(e) ; list_of() = l ^ ⟨e⟩ end
    end,
```

2.3 Execution of Tests

Tests are executed using the implementation of the candidate component. Thus, we need to rename the functions using the identifiers of the respective methods offered by the component. For example, the test function based on the axiom [list_of_enq] is translated into C++ as:

```
...
Bool test_list_of_enq (const Term code,
            const IntList parameters, const int e) {

    intRSList* _listQueue;
    CPP_Queue* _queue;
    Bool result ;

    _queue->generate(code, parameters);
    if (_queue->is_full()) {
      result = true;
    }
    else {
      _listQueue = _queue->list_of();
      _queue->enq(e);
      result = _queue->list_of() == _ilistQueue +
               intRSList(e, intRSList ());
    }
    return result;
}
...
```

The type of interest is translated as CPP_Queue. An object of type CPP_Queue is created (_queue) and then it is updated according with the information included in the parameters ($_queue- > generate(code, parameters)$). The parameters *code* and $parameters$ indicate which generators should be used to build a valid term to instantiate the test. For example, the actual parameters "mee" and 5,8 denote the term $enq(enq(5, empty()), 8)$. Note that the function $list\_of$ is hidden (see A_QUEUE0), but we need to implement it to execute the tests.

Suppose we want to evaluate if the MFC Library[1] component p_queue behaves as desired. We should adapt this code to replace the definitions of the type Queue with the definition of the class p_queue. As an illustration consider the following portion of code:

[1] Microsoft Foundation Class Library

```
...
Bool test_int_list_of_enq (const Term code,
                const IntList parameters, const int e) {

    typedef priority_queue<Data,DataVector,less<Data>,
                allocator<Data>> DPQueue;

    intRSList* _listQueue;
    DPQueue* _pQueue;
    Bool result ;

    _pQqueue->generate(code,parameters);
    if (_pQueue->is_full ()) {
      result = true;
    }
    else {
      _listQueue = _pQqueue->list_of();
      _pQueue->push(e);
      result = _pQueue->list_of() == _listQueue +
               intRSList(e,intRSList());
    }
    return result;
}
...
```

We replace _queue with the object of type DPQueue _pQqueue. We should extend the class DPQueue with the new methods *list_of* and *generate*. In order to keep this presentation as simple as possible we have supressed some implementation details.

As a result of the execution of tests we get a true result if the test is successful, otherwise, the output is false. For the case of the [list_of_enq] axiom we get a false output when we enqueue more that one item. The pop method inserts items in decreasing order while the expected behavior is of a traditional queue. For example, the test with the term $enq(enq(5, empty()), 8)$ fails because the object contain the elements "8,5", but we expect to have a queue with elements "5,8".

3 Conclusions

We have described how to use the testing approach reported in [9] for the case of component assesment and developments specified using an algebraic language. The approach can be easily adapted for

systems specified using a Hilbert model, in [10] we describe the generation of test cases for systems specified using this model.

The main features of our proposal are the use of a semantic criteria to assess components, and that it can be applied only when the implementation of the component is available. We do not assume the existence of a formal specification of the candidates components because this assumption is quite restrictive. Then, our approach is of wide interest in the software engineering community.

Automated retrieval of software components is usually achieved with *text-retrieval* methods [11], [12]. These methods search for words or phrases associated with a component. There are also controlled term vocabularies approaches based on classification schemes which are used to structure libraries [13]. Other retrieval systems use artificial intelligence techniques to represent and reason about knowledge of component semantics [14]. However, considerable manual domain analysis is required to build powerful retrieval systems using these techniques.

Podgurski *et al.* [15] proposed an approach for selecting and testing reusable components that is very close to dynamic assessment. The authors propose a *behavior sampling* that identifies relevant routines by executing candidates on a searcher supplied sample of operational inputs and comparing their output to output provided by the searcher. Although this work is very close in spirit to ours, they do not address in a systematic way the problems of definition of tests (which tests should be executed), selection of a finite set of terms, and, more importantly the oracle problem.

One important limitation of our approach is that we cannot retrieve components whose behavior is close, but not identical, to that required. In order to address this limitation we should define a theory for the behavior that an adapter must provide, such that, when composed with the candidate component we obtain the required behavior. In particular, if the candidate component has missing behavior with respect to the specification, we need to include in the adapter definitions of new functions. Also, more axioms will be required to explore the interactions among all funcions. This issue requires thorough investigation. We intend to work on this topic in the future.

References

1. Alan W. Brown. *Large-Scale Component-Based Development*. Prentice Hall International, 2000.
2. David Krieger and Richard M.Adler. The emergence of distributed component platforms. *IEEE Computer*, March:43–53, 98.
3. A. Zaremski and J. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4), 1997.
4. The RAISE Language Group. *The RAISE Specification Language*. The Practitioner Series. Prentice Hall International (UK) Limited, 1992.
5. The RAISE Method Group. *The RAISE Development Method*. The Practitioner Series. Prentice Hall International (UK) Limited, 1995.
6. P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3), june 1993.
7. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.
8. Bruno Marre. Toward automatic test data set selection using Algebraic Specifications and Logic Programming. In *Proceeding of the Eight Internationl Conference on Logic Programming, ICLP91*. MIT Press, 1991.
9. Marisa A. Sanchez. Integrating Testing with a Formal Development Process. In Ram Chillarege and Thomas Illgen, editors, *ISSRE 98, The Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, November 4-7 1998*, pages 205–213, 1998.
10. Marisa A. Sanchez and Juan Carlos Augusto. Testing an Implementation of a Temporal Logic Language. In *2000 20th International Conference of the Chilean Computer Society, Santiago, Chile, 13-18 Nov 2000*. IEEE Computer Society Press, 2000.
11. G. Salton. Another look at automatic text-retrieval systems. *Communications of ACM*, 29(7):648–656, July 1986.
12. S. P. Arnold and S.L. Stepoway. The REUSE system: Cataloging and retrieval of reusable software. In *Proceeding of the 1987 Spring Joint Computer Conference, San Francisco, February 23-27, 1987*, pages 376–379. IEEE, 1987.
13. R. Prieto-Diaz. Implementing faceted classification for software reuse. In *Proceedings of the 12th International Conference on Software Engineering*, pages 300–304. IEEE Computer Society Press, 1990.
14. H. Tarumi, K. Agusa, and Oiino. A programming environment supporting reuse of object-oriented software. In *Proceedings of the 10th International Conference on Software Engineering*, pages 265–273. IEEE Computer Society Press, 1988.
15. A. Podgurski and L. Pierce. Retrieving Reusable Software by Sampling Behavior. *ACM Transactions on Software Engineering and Methodology*, 2(3):286–303, July 1993.