ECOOP
MÁLAGA·SPAIN
2002

WITUML: Workshop on Integration and Transformation of UML models

Málaga, Spain, June 11th, 2002

Organizing Committee

- *João Araújo: Universidade Nova de Lisboa, 2829-516 Caparica, Portugal, ja@di.fct.unl.pt*
- *Jon Whittle: NASA Ames Research Center, Moffett Field. CA 94035 , jonathw@ptolemy.arc.nasa.gov*
- *Ambrosio Toval: Universidad de Murcia, Murcia, Spain, atoval@dif.um.es*
- *Robert France: Colorado State University, Fort Collins, CO 80523 , france@cs.colostate.edu*

Accepted Position Papers

- From UML to Ptolemy II simulation: a model transformation, V.Arnould, A.Kramer, F.Madiot
- Generating test data from OCL specification, M.Benattou, J.-M.Bruel, N.Hameurlain
- Adapting design types to communication media and middleware environments, J.Cañete, F.Galán, M.Toro
- Mapping models between different modeling languages, E.Domínguez, A.Rubio, M.Zapata
- Using temporal logic to represent dynamic behavior of statecharts, M. Enciso, I. Guzmán, C. Rossi
- Generating and executing test data f or OO systems, M. Jiménez, M. Polo, M. Piattini
- Integration of UML models using B notations, H. Ledang, J. Souquières
- OCL as a core UML transformation language, D. Pollet, D. Vojtisek, J.-M. Jézéquel
- A framework for model transformations, I. Porres
- Mapping OO applications to relational databases using MOF and XMI, E. Rodrigues, R. Melo, F. Porto, J. Neto
- The value added invariant: a newtonian approach for generating class diagrams from a use case model, B. Roussev
- Adding use cases to the building of information systems with Oracle Designer, P. Saliou, V. Ribaud
- A pragmatic approach to rule-based transformations within UML using XMI.difference, A. Wagner

Generating and executing test data for objects oriented systems[1]

María del Mar Jiménez, Macario Polo, Mario Piattini
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4; 13071-Ciudad Real (Spain)

Abstract

This paper presents the first research results of a method for the automatic generation and execution of test cases for Object Oriented Systems.

The starting-point is the algebraic modelling of the static structure (class diagrams) of the object-oriented system and, then, the application of a set of algebraic transformation functions to obtain the required test cases.

1 Introduction.

Software testing involves judging how well a series of test inputs test a piece of code [7]. Manual generation of such tests can be quite time consuming, and software testing has became one of the most expensive stages of the software life cycle [9]. So, great research effort is being devoted to automate its tasks, as test-case generation and test-case execution [6].

In this paper we present the first results of a method for the automatic generation and execution of test cases in object-oriented systems. We have defined an initial metamodel that, using algebraic notation, allows to represent the class structure of any object-oriented system. Then, the object-oriented system can be translated into an instance of such metamodel and, from the application of a set of algebraic functions working on it, a wide number of test cases can be obtained.

This very same metamodel is also valid for doing other software engineering tasks, as code generation [8] or reverse-engineering.

This position paper is organized as follows: in Section 2 we briefly present the structure of the metamodel used, and the way used to convert an object-oriented system in one instance of the metamodel; in Section 3 we examine the most common techniques of test case generation and some problems values for these test case. Finally, in section 4 we describe how executing the test data.

2 Metamodel description.

Our metamodel, in its current point of development, specifies class diagrams using a formal representation based on mathematical descriptions of many of the elements existing in an object-oriented system.

There are many proposals related to the formal specification of object-oriented systems, such as TROLL [5], Object-Z [11] or even OCL [4]. Although probably our ideas could be put into practice in any of these environments, the reality is that these proposals are too rigorous ("too formal") to popularise its use. Our approach is closer to the ideas of Manfred Broy [3], for whom Software Engineering, as any other discipline, needs mathematical descriptions and theories for its modeling aspects,

[1] This work is partially supported by the DOLMEN project (CICYT, TIC-2000-1673-C06-06).

description techniques and development methods. However, he claims that such mathematical theory must not be too complex, since then it would not be a very helpful contribution for software engineers. The theory should give semantics to usual description techniques (as class diagrams, state charts, etc.) and should explain methods from a mathematical rational, more in the form of a "Formal Description Technique" than as a "Formal Method". This involves the formal description of usual software-engineering techniques through usual mathematics.

With this consideration, a possible textual description of the static structure (class model) of an object-oriented system would say that a class model is a set of classes and relations between them: associations, aggregations, dependence and inheritance relationships. From this, a formal description of a class model is a "system" as defined by [7]: it is composed of classes and relationships among them:

$S=(C, R)$, where C is the set of classes and interfaces and R is the set of existing relationships among elements in C: $R \subseteq C x C$.

A class c is an element of C; this is: $c \in C=(Name, Fields, Constructors, Methods, Parents)$, where *Fields* is the set of the class fields; *Constructors* is the set of operations that allow to build instances of c; *Methods* is the set of its methods; *Parents* is the set of parent classes ($Parents \subseteq C$).

Each one of these elements can be described in more depth, giving details on the definition of fields, constructors and methods:

- $f \in Fields=$ *(Name, Class, Visibility)*, where *Name*, *Type* and *Visibility* respectively are the name, class, and visibility (public, etc.) of the field.
- $t \in Constructors=$ *(Name, Visibility, Arguments)*, where *Name* is the same name of the class and *Arguments* is the set of the arguments of this constructor, being $a \in Arguments=(Name, Class)$, where *Class* is the type of the argument.
- $m \in Methods=$ *(Name, Class, Visibility, Arguments, isStatic)*, where *Class* is the type of the result returned by the method, and $isStatic \in \{true, false\}$ denotes whether the method is or not a class method.

With these descriptions, the set of classes and relationships in an object-oriented system remains as follows:

$S=\{C,R\}=$
$\{(Name_1, Fields_1, Constructors_1, Methods_1, Parents_1),$
$(Name_2, Fields_2, Constructors_2, Methods_2, Parents_2),$
$\ldots,$
$(Name_n, Fields_n, Constructors_n, Methods_n, Parents_n),$
$R_1, R_2, \ldots, R_n\}$

Therefore, any object-oriented system, written in any programming or modelling language, could be described using this single algebra. It is possible to define transformation functions on these sets to perform many software engineering tasks. Test-case generation and execution are two of them.

3 Test case generation.

There are a number of techniques for automatically generate and execute test-cases, such as:

- Random Test Data Generation, that generates random values to be used as inputs of the program under test.

- Symbolic Execution, that makes an static analysis of the source code to generate test cases and simulates the actual execution of the program [6]. Symbolic execution is difficult to use in real environments, due to its drawbacks to deal with pointers, arrays, complex data types, etc.
- Dynamic Test Data Generation, that tracks the result of the program execution to progressively generate test-inputs that help to reach higher values of coverage [7].

3.1 Algorithm

The test of a class consists of the construction of an instance of the class (via the invocation of one of its constructors) and the execution, on that instance, of a sequence of some of its methods. The selected constructor and methods can take parameters, that can be of any type. Supposing a constructor or a method that takes an integer as parameter, it could be invoked using random values, boundary values (close to zero), very big and very small values, etc. as test-data. If another operation takes a String as parameter, it could be invoked using the empty String, the null string, a string of special characters, a string of letters and numbers, etc. as test-data. If there is another operation that takes an integer and a String as parameters, it could be invoked with all the combinations of all the afore-mentioned values. When the parameter is not a single data-type (its type is a complex class, as "Person", "Car", etc.), it is possible to generate instances-for-test combining the test values of its fields.

Our algorithm is based on the previous idea: it starts generating test cases based on the types of the fields of the class. For example, given a class *c* with two fields, one *integer* and one *String*, we can generate so many test data as combinations of predefined values for these types. Supposing we have predefined {-1, 0, +1} and {new String(), null, "Hello!"} as basic values for testing integer and string variables, we would have the following combinations for constructing instances-for-testing of *c*:

• (-1, new String())	• (-1, null)	• (+1, "Hello!")
• (0, new String())	• (0, null)	• (0, "Hello!")
• (1, new String())	• (1, null)	• (+1, "Hello!")

This is, we define a set of predefined test-data for basic types, and then we use the cross product of the fields in the class to generate test-data in classes with more complexity. These test-data are used as values for invoking constructors and methods of the class-under-test. More formally, being *TD* the test-data for a class:

$$TD\ (Primitive\ Type) = \{Predefined\ Values\}$$

$$TD\ (t \in Constructors) = \{\prod_{\forall a \in Arguments} TD\ (a.Class)\}$$

$$TD\ (m \in Methods) = \{\prod_{\forall a \in Arguments} TD\ (a.Class)\}$$

Finally, we obtain a test case for a class invoking a constructor of this class and a collection of methods, and passing as parameters some pre-generated values:

$$TC\ (c \in Classes) = (t \in Constructors,\ m \subseteq c.Methods)$$

3.2 Example

Let us consider the following three classes: Person, Employee and Student, as in Figure 1.

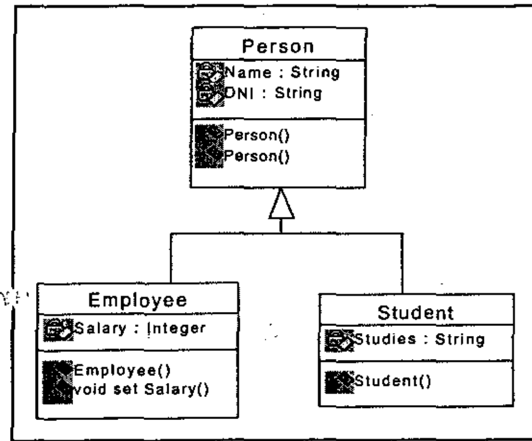

Figure 1. Example Class Diagram.

According to section 2, the algebraic representation of these system is:

$S=\{c_1, c_2, c_3, r_1, r_2\}$
$c_1=(Person, \{f_1, f_2\}, \{t_1, t_2\})$
$f_1=(DNI, String, Private)$
$f_2=(Name, String, Private)$
$t_1=(Person, Public, \{a_1, a_2\}, True)$
$a_1=(DNI, String)$
$a_2=(Name, String)$
$t_2=(Person, Public, \{p_1\}, True)$
$a_1=(DNI, String)$
$c_2=(Employee, f_1, t_1, m_1, Person)$
$f_1=(Salary, Integer, Private)$
$t_1=(Employee, Public, \{a_1, a_2\}, True)$
$a_1=(P, Person)$
$a_2=(Salary, Integer)$
$m_1=(setSalary, void, Public, a_1, True)$
$c_3=(Student, f_1, t_1, Person)$
$f_1=(Studies, String, Private)$
$t_1=(Student, Public, \{a_1, a_2\}, True)$
$a_1=(P, Person)$
$a_2=(Studies, String)$

Once we have a formal description of the classes, we can generate the data test. Let us suppose we already have the following set of predefined values for primitive data types (such as zero, empty string, random values, etc.):

TD(Integer) = {-1820, -7, -1, 0, 1, 52, 2090} // 7 values
TD(String) = {"", NULL, "a", "1", "#", "$", "Ann", "eeeeeeehhhh"} // 8 values

In order to test, for example, the class Person, we need first to define a set of test sequences, composed by a constructor and some methods. The first constructor, for example, that takes two strings as parameters, what would produce 8*8=64

combinations of values for testing this constructor. If the class would have methods, a similar mechanism would be used to generate test-data for invoking the methods.

When a test-sequence is executed, it produces an object that can be saved for doing later regression testing. In a real environment, this is got via Serialization.

4 Conclusions and future work.

This paper has presented the first results of a method for the automatic generation of test-cases in object-oriented systems. The main idea is to use a metamodel for representing any object-oriented system and, then, the application of algebraic functions working on such metamodel to generate the desired test cases.

We are currently working on the refinement of the metamodel to support the characteristics of most of the object-oriented programming languages, and in the development of tools to translate object-oriented programs and class diagrams depicted with commercial CASE tools into a common notation that represents instances of the metamodel. This common notation is physically represented as XML files.

An important future line of work is the development of notation and algorithms for representing and executing testing sequences from the generated test cases.

5 References.

[1] Biggerstaff, TJ., Mitbander, BG. and Webster, DE. (1994). Program Understanding and the Concept Assigment Problem. *Communications of the ACM*, 37(5), 72-83.

[2] Briand, L. C., Morasca, S. and Basili, V.R. (1996). Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering*, 22(1), 68-86.

[3] Broy, M. (2001). Toward a Mathematical Foundation of Software Engineering Methods. *IEEE Transactions on Software Engineering*, 27(1), 42-57.

[4] Heinrich H., Frank F., Ralf W. Using Previous Property Values in OCL Postconditions - An Implementation Perspective. *UML 2.0 - The Future of the UML Constraint Language OCL*, October 2, 2000, York, UK.

[5] Jungclaus R., Saake G., Hartmann T., and Sernadas C. (1991). Object-Oriented Specification of Information Systems: The TROLL Language. *Informatik-Bericht* 91(04).

[6] Meudec C. ATGen: automatic test data generation using constraint logic programming and symbolic execution. *Journal of Software Testing, Verification and Reliability* 2001; 11(2):81-96.

[7] Michael, CC., McGraw, G., Schatz MA. (2001). Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27(12), 1085-1109.

[8] Polo M, Mayoral A, Gómez JA and Piattini M. (2001). *Automatic generation of fully-executable code from the Domain tier of object-oriented systems.* Proc. of the First workshop on Transformations of the Unified Modelling language (WTUML). Genove, Italy.

[9] Pressman RS. *Software Engineering: a practitioner's approach.* McGraw-Hill, 1997.

[10] Reinder J., Loe M. G Feijs, André Glas, René L. Krikhaar, Thijs Winter (2000). Maintaining a legacy: towards support at the architectural level. *Journal of Software Maintenance*, 12(3): 143-170.

[11] Smith G. The Object-Z Specification Language. Kluwer Academic Publishers, 2000.