Testing Techniques for a Logic Representation Generator

Ciprian-Bogdan Chirilă
Politehnica University of Timişoara
ciprian.chirila@cs.upt.ro

Călin Jebelean
Politehnica University of Timişoara
calin.jebelean@cs.upt.ro

Krisztina Francz
kfrancz@gmail.com

Abstract

Logic based representation can be used for expressing programs and models driven by a grammar. Thus, model analysis and transformation written as declarative paradigm rules can be more expressive. Usually, logic representations are obtained by translators which must be tested as any other software artifacts. We present several testing techniques in the context of logic based representation.

1 Introduction

Our work is based on the logic foundation for program representation defined in [11]. The logic representation may be seen as a set of Horn clauses organized as a 3NF normalized relational database having no null values. In practice the Horn clauses are implemented as Prolog clauses. Logic representation has two main uses for programs and models: analysis and transformation. These uses can help in achieving several other goals in the programming world. Using logic representation for programs allows writing analysis and transformation rules using declarative languages. Thus, these rules are more expressive than writing them using other paradigm like the one of imperative languages. Another major advantage of the approach is that the logic based representation is language independent. We experimented it on Java [2] and Eiffel [13]. Currently, logic representation is used in the context of object-oriented technology for several purposes like: i) the detection for the lack of design patterns in object-oriented design [9]; ii) the detection and extraction of concerns in the context of separation of concerns paradigm [12]; iii) the implementation of the reverse inheritance class relationship semantics for Eiffel in order to favor code reuse and to facilitate "a posteriori" architecture redesign [4, 5]. The main diagram of our logic based framework is presented in figure 1. The first rectangle block of the framework is the logic representation generator

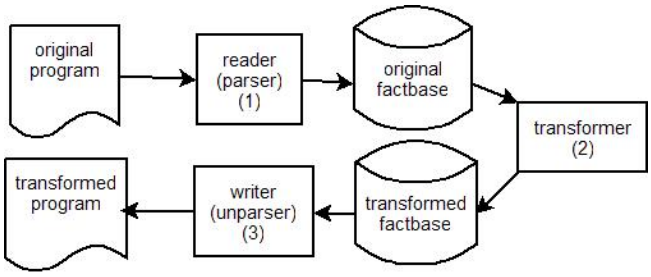

Figure 1. Logic Based Approach

which translates the input program into logic representation or factbase. In the remaining of the paper we will refer to this parser as **reader**. In [10] it is shown that for any model expressed through a grammar it can be automatically generated a parser which can output a logic representation of that model. Otherwise the **reader** must be written manually. The second rectangle block is a transformer which can extract information from the factbase or can change the model by adapting, evolving or enhancing it. The third rectangle block is the writer which unparses the resulted factbase and regenerates the program. In this paper we present several black box testing techniques [3] for a test automation framework to assist the programmer in the development process of the **reader**. Some of the used techniques are classic and some have some original elements. The paper is structured as follows: in section two we present the main concepts of the logic based representation; in section three we describe several testing strategies for the generator; in the fourth section related works are presented; finally, in section five we conclude and we set the perspectives.

2 Logic Based Representation in a Nutshell

In this section we will briefly present the basics of the logic based representation which was defined in [11] for Java. We will illustrate these concepts through an example in the Java programming language. The approach on logic-based representation we use is language independent, as was shown in [10]. Figure 2 presents a simple example

of a Java class *Person*, with a field representing the person's age and a setter for this field. In figure 2 we have the exam-

```
01 class Person {
02  int age;
03  void setAge(int age) {} }
04
05 classDecl(100,0,'Person',[101,102]).
06 fieldDecl(101,100,'age',int).
07 methodDecl(102,100,'setAge',void,[103],104).
08 param(103,102,'age',int).
09 block(104,102,[]).
```

Figure 2. Logic Representation for a Java Code Example

ple Java code between lines 01-03 and the equivalent logic representation as Prolog knowledge between lines 05-09. As one can easily notice, the Prolog model contains information that's already present in the Java code, only in a hierarchical and logic fashion. Each Prolog fact refers to a Java entity. These entities are also linked in Prolog by means of numeric identifiers. Thus, each entity has a personal and unique numeric ID as first parameter and the numeric ID of its ancestor (father) as second parameter. The other parameters (if available) offer additional information about that particular entity. The first two parameters of each Prolog fact express a father-son relation between entities. Thus, fields and methods are linked to the class where they belong, parameters and blocks are linked to the method to whom they belong, etc. The whole knowledge base becomes a big generalized tree if only the first two parameters are considered from each fact. This tree is, in fact, the abstract syntax tree of the program under analysis. In the example from figure 2, the first Prolog fact (line *05*) states that 'Person' is the name of a class, its ID is 100 and its parent ID is 0 (this means it has no parent). The last parameter is a list of IDs representing the IDs of the class members. We can see there are two members in the class, having 101 and 102 as their personal IDs. If we further inspect the knowledge base, we can see that 101 is the ID of a field called 'age' of type int, while 102 is the ID of a method called 'setAge' of type void, having one parameter with ID=103 (there could be more parameters, that's why we use a Prolog list to hold them) and one method block with ID=104. The modeling process is exhaustive, going all the way down to the instruction level and even beyond, to the atomic level. For reasons of simplicity, we limited the example to the method block level, but instructions within each method block would have been considered and modeled further as sons of the method block fact. Next sections deal with a few techniques aimed at testing the consistency of the parser that generates such logic knowledge bases.

3 Parser Testing Techniques

In this section we present an overview of the **reader** parser we are going to test. The **reader** transforms programs written in a certain programming language into logic based representation. The **reader** may be language independent but some tests are language dependent and we will illustrate the testing strategies on Eiffel programming language. In our test plan we designed several test scenarios, each scenario tests a certain language entity: class header, inheritance branches, features, types, instructions and expressions. Each scenario is tested using several techniques. We will illustrate our testing strategies on one example dealing with feature blocks, features and client classes. We consider the following input Eiffel code from figure 3 lines 01-06. The correct expected output is listed in the same figure between lines 08-15.

```
01 class FIGURE
02  feature {COMPOUND_FIGURE, DRAWING}
03   draw is do ... end
04 end
05 class COMPOUND_FIGURE ... end
06 class DRAWING ... end
07
08 cluster(10,'.').
09 classDecl(100,10,'FIGURE').
10 classDecl(101,10,'COMPOUND_FIGURE').
11 classDecl(102,10,'DRAWING').
12 featureBlock(200,100).
13 featureDecl(300,200,'draw').
14 featureClientClass(400,200,101).
15 featureClientClass(400,200,102).
```

Figure 3. Logic Representation for an Eiffel Code Example

Output Testing Output testing is a black box testing technique where we set a certain input and we expect a certain output. For such a test the input is an Eiffel program and the output is a Prolog factbase. The comparison is made with the **diff** [7] tool between the output file and the oracle file. The test succeeds if the two files are identical. For example, for the input in figure 3 we expect to get the output from the same figure. The advantage of such a technique is that the comparison is very strict. Any modification in the input sample will determine changes in the output, thus triggering test failure. The disadvantage stands in the fact that even some minor modifications in the sample, even ones not related to the tested entity, will trigger test failure. For example, if the identifiers of facts are changed due to some fact generation order change, the test will detect false positives. The creation and maintenance of such a test battery is done manually but it is used as regression test automated by **Perl** [1] scripts and by the **diff** [7] tool.

Logic Testing Logic testing is also a black box testing technique where we test the existence of several facts and the relations between them, within the output factbase. These tests can be easily constructed by: i) inspecting manually the output facts; ii) selecting the desired facts which are related; iii) changing automatically the identifiers into

anonymous variables; iv) adding the obtained rules to the oracle of the test case collection. For example, in the case of Eiffel language the facts refering to features, feature blocks and client classes are described in figure 4. The generated

```
featureBlock(#id,#classDecl).
featureDecl(#id,#featureBlock,'FeatureName').
featureClientClass(#id,#featureBlock,#classDecl).
```

Figure 4. Metamodel Fragment

output facts are listed in figure 3. From this list the programmer may easily select the facts from figure 5. The re-

```
01 featureBlock(200,100).
02 featureDecl(300,200,'draw').
03 featureClientClass(401,200,101).
04 featureClientClass(402,200,102).
```

Figure 5. Selected Facts

lations which can be inferred automatically from the facts of figure 5 are listed in figure 6. The rule is inferred by

```
01 featureBlock(_200,_100),
02 featureDecl(_300,_200,'draw'),
03 featureClientClass(_401,_200,_101),
04 featureClientClass(_402,_200,_102).
```

Figure 6. Inferred Rule

changing the facts primary and foreign keys into anonymous variables and coupling them with the **and** operator. The anonymous variable is created by adding the "_" (underscore) symbol as prefix to the integer keys. This rule will be used to test the generated output. If some fact arguments are atoms we can choose to make or not to make them anonymous variables. Leaving atoms as such, will make the rule more particular for a certain language entity present in the factbase. For example in line *02* of figure 6 the rule will refer only to the *draw* feature from the factbase. Otherwise, the rule will check for any relation in the generated factbase, regardless of any feature name. The drawback of this technique is that the creation of such a test battery is done manually but the advantage is that no maintenance is needed during the parser development process unless the metamodel is changed.

Type Testing Because facts are linked by primary and foreign keys and because they are globally unique, we can check if the referred fact types correspond. Such a verification is equivalent to the type checking stage of a regular compiler. For example for the facts in figure 4 we must check that: i) the second argument of the *featureBlock* fact refers a *classDecl* fact; ii) the second argument of *featureDecl* fact refers a *featureBlock* fact; iii) the second argument of *featureClientClass* refers a *featureBlock* fact; iv) the third argument of *featureClientClass* refers a *classDecl* fact. Such a verification could be made automatically by a metaroutine if there is a formal description of the metamodel. In figure 7 we present the type checking rules for *featureDecl* and *featureClientClass* facts. Rule *checkFeatureDecl* checks that all arguments of the analysed fact correspond to a fact of a certain type meaning that *FeatureBlockId* identifies a feature block. Next, all the client class facts of the current fact are type checked. For this purpose we call the *checkFeatureClientClassRule* which will check that the second argument of the fact is a feature declaration and that the third argument represents a class declaration.

```
checkFeatureDecl(FeatureDeclId):-
 exists(featureDecl(FeatureDeclId,
  FeatureBlockId,_FeatureName)),
 exists(featureBlock(FeatureBlockId,_)),
 forall(exists(featureClientClass(
  FeatureClientClassId,_,_)),
  checkFeatureClient(FeatureClientClassId)).
checkFeatureClientClass(
  FeatureClientClassId):-
 exists(featureClientClass(
   FeatureClientClassId,
  FeatureDeclId,ClassDeclId)),
 exists(featureDecl(FeatureDeclId,_,_)),
 exists(classDecl(ClassDecl,_,_,_)).
```

Figure 7. Type Checking Rule

4 Case Study: ETransformer

The presented testing techniques were successfully applied on the Eiffel to Prolog translator named **ETransformer**. This tool is used in the implementation framework for the semantics of **reverse inheritance** class relationship of Eiffel [5]. For each entity of the language we wrote an input Eiffel code sample. The tested entities are: class header, inheritance and exheritance branches, creator declarations, feature blocks, features, types (class, expanded, separate, like, bit), formal arguments, locals, assertions, instructions (creation, assignment, conditional, multibranch, loop, retry, debug, call) and expressions (subexpression, unary operator, binary operator, manifest constant, call). We built around 220 test cases.

5 Related Works

The functional testing or black box testing defined in [3] explains the principles behind behavioral testing: the system is tested based on its desired behavior and for conformance to its specifications. Our work uses this principle in the output testing and logic testing techniques. In the former we compare the output with an oracle and in the latter we check the existence of expected facts. The first testing technique uses the black box principles as such while the second testing technique allows the creation of tests derived from the structure of the input sample.

[8] defines an automatic process of generating model based tests, taking as input a formal model of the tested software and a set of generation directives. A test generator must output a set of test cases that include a sequence

of inputs and the expected responses of the system. Next, a series of test generators are presented. Our work belongs to this category only in what concerns the first method (output testing). However, it is closely related to most of the frameworks presented there in what concerns automation of the test generation process.

[14] presents some graph based techniques for model testing. The process is based on modeling the states of software as nodes in a graph and the actions that can be taken in each state as links between nodes. Then, several graph traversal algorithms are applied to test the software system.

[6] describes a structural testing technique, the code coverage analysis on several statements like decisional statements, calls, loops. This analysis deals with finding areas of programs not exercised by the testcases. This technique allows to identify redundant test cases that do not increase code coverage.

6 Conclusions and Perspectives

In this paper, we have presented some techniques to test the generation process of logic representation for programs. We currently have a suite of generators of simple logic representation for any programming language that were automatically generated and a manually-written generator of logic representation aimed at programs written in the Eiffel language. While the automatically obtained generators were quite unlikely to contain errors, the manually written one would obviously benefit from a methodology aimed at testing its consistency.

Logic representation generators are not different than any other software artifacts, so all common testing techniques may apply to them. Furthermore, due to the special kind of output we showed that some other particular testing techniques may be used. Output testing needs manual validation of the output. Logic testing is semiautomatic and implies manual selection of facts, but afterwards the testing process is automatic. Type testing is a fully automatic process if a formal metamodel is given. Logic testing apparently overlaps type testing, but logic testing "expects" some facts while the type tester checks the types of existing facts. In other words, if the reader does not provide the "expected" facts the type tester may still validate the output. As an immediate perspective, using the metamodel of the facts, we could develop a generic routine to test the type of the arguments with the facts which are referred. Another perspective would be to use these testing techniques on other readers like those for Java or on a generic reader.

References

[1] Perl - Practical Extraction and Report Language. http://www.perl.org, 2009.

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.

[3] Boris Beizer. *Black Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley and Sons, New York, USA, 1995.

[4] Ciprian-Bogdan Chirila, Pierre Crescenzo, and Philippe Lahire. Reverse inheritance: Improving class library reuse in Eiffel. Langages et Modeles a Objets 2007 (poster), Toulouse, France, May, 2007.

[5] Ciprian-Bogdan Chirila, Günter Kniesel, Philippe Lahire, and Markku Sakkinen. The Eiffel/RI Project Website. https://nyx.unice.fr/projects/transformer, 2009.

[6] Steve Cornett. Code coverage analysis. http://www.bullseye.com/coverage.html, 2008.

[7] Paul Eggert. Diffutils. http://www.gnu.org/software/diffutils/diffutils.html, 2008.

[8] Alan Hartman. AGEDIS Model based test generation tools. http://www.agedis.de/documents/ ModelBasedTestGenerationTools_cs.pdf.

[9] Călin Jebelean. Automatic detection of missing abstract-factory design pattern in object-oriented code. In *Proceedings of the International Conference on Technical Informatics*, University Politehnica Timişoara, 2004.

[10] Călin Jebelean, Ciprian-Bogdan Chirila, and Anca Măduţa. Generating logic based representation for programs. In *In Proceedings of 2008 IEEE 4-th International Conference on Intelligent Computer Communication and Processing*, pages 145–151, Cluj-Napoca, Romania, August 2008.

[11] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Dept. III, University of Bonn, January 2006.

[12] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Workshop on Linking Aspect Technology and Evolution (LATE'07), Vancouver, British Columbia*. March 2007.

[13] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/˜meyer/, September 2002.

[14] Harry Robinson. Graph theory techniques in model-based testing. In *In Proceedings of the International Conference on Testing Computer Software*, 1999.