Test Case Generation for Programming Language Metamodels

Abstract for Software Language Engineering 2010 Doctoral Symposium

Hao Wu*
Supervisors: Rosemary Monahan and James F. Power

Department of Computer Science, National University of Ireland, Maynooth
{haowu,rosemary,jpower}@cs.nuim.ie

1 Problem Description and Motivation

One of the central themes in software language engineering is the specification of programming languages, and domain-specific languages, using a *metamodel*. This metamodel provides a greater degree of abstraction than a context-free grammar, since it can ignore syntactic details. However, its main benefit is in providing for the specification of the abstract syntax graph of a language, and this can then be used for the construction or generation of language processing tools.

One problem associated with the use of programming language metamodels, and metamodels in general, is determining whether or not they are correct. Of course, one approach is to forward engineer code and test this, but it should also be possible to test the metamodel directly. In this context, the question addressed by our research is: *given a programming language metamodel, how can we generate an appropriate test suite to show that it is valid?*

Being able to generate such a test suite would have two main benefits. First, examining the automatically-generated test cases would help to develop the modeller's understanding of the metamodel, and help to increase confidence in its validity. Second, since the metamodel specifies a programming language, the generated test cases should be valid programs from that language, and these can be used as test inputs for tools that process the language.

2 Related work

Testing a programming language specifications, at least at the syntactic level, has a long history, dating back at least to the work of Purdom on generating test cases from grammars [**?**]. However, a naive application of Purdom's approach to a programming language grammar produces programs that may not be syntactically correct (since the grammar may under-specify), and is certainly unlikely to produce semantically valid (let alone meaningful) programs [**?**].

* This work is supported by a John & Pat Hume Scholarship from NUI Maynooth.

Incorporating at least a language's static semantics into test suite generation must go beyond simple context-free grammars. One possible approach is to use attribute grammars, and to seek to exploit the attribute equations to constrain or even direct the generation of test cases [**?**,**?**]. Despite this research, it is still not a trivial task to directly extend this work to a metamodelling environment that uses, for example, OCL-like constraints for describing the static semantics.

It is possible to borrow some ideas from software modelling, and there is a great deal of existing research dealing with model-based testing strategies [**?**,**?**]. However, much of this work focuses on the behavioural elements of software models (such as state machines), rather than the structural aspects which might be more relevant to metamodelling.

In the context of testing structural models, two tools strike us as being particularly noteworthy:

- **USE** (a UML-Based Specification Environment) which allows the user to specify a UML (Unified Modeling Language) class diagram with OCL constraints [**?**]. From these we can manually generate corresponding object diagrams, and the USE environment will check that these are valid instances, satisfying the relevant class invariants.
- **Alloy** which has its own logical specification language, corresponding roughly to the elements found in a class diagram, and automatically generates object diagrams that correspond to this specification [**?**]. One of the useful aspects of the Alloy tool is that it is designed to work with a number of different SAT solvers in order to test constraints and generate counter-examples.

As part of an earlier project, we have previously exploited the close relationship between the Alloy notation and UML class diagrams to generate instances of a metamodel for software metrics [**?**]. One of the drawbacks of this earlier approach is that it did not control the strategy used to generate instances. Faced with the impossibility of manually validating several hundred thousand instances, we exploited test-suite reduction techniques to narrow the set of test cases.

3 Proposed Solution

It is clearly inefficient to generate test cases that are not used, and an ideal solution would be to generate an appropriate set of test cases in the first place. Of course, this immediately raises two questions: what do we mean by an *appropriate* set of test cases, and how do we generate these test cases?

One way of measuring the adequacy of a set of test cases for a piece of software is to use coverage criteria. Typically, a test suite can be judged in terms of the percentage of statements, decisions, branches etc. in the source code that are executed when the software is run. It seems natural therefore to attempt to seek to assemble a test suite for a metamodel along similar lines, i.e. to form a set of models that cover the features of the metamodel. In terms of programming language metamodels, this would involve creating a set of programs that exercise the features of the language.

Since most metamodels, including programming language metamodels, are described using UML, it is possible to use UML techniques to compute their coverage. Many coverage criteria for the various UML diagrams have been proposed [**?**,**?**,**?**,**?**]. However, we restrict our attention to UML structural diagrams, since metamodels are often presented in the form of UML class diagrams.

Our recent work has focused on coverage criteria for UML class diagrams initially proposed by Andrews et al. [**?**], specifically:

- Generalisation coverage which describes how to measure inheritance relationships.
- Association-end multiplicity coverage which measures association relationships defined between classes.
- Class attribute coverage which measures the set of representative attribute value combinations in each instance of class.

It is unlikely that these alone will provide sufficient granularity to determine the adequacy of a test suite, and previous work has already determined that there is a poor correlation between coverage of syntactic and semantic features [**?**]. However, our immediate work has centred on constructing an extensible, modular system that can at least measure these levels of coverage, and which can be used as a basis for further studies.

4 Research Method

Our research can be broken down into three phases:
Phase I: calculating coverage measures for programming language metamodels,
Phase II: generating valid models that satisfy coverage criteria,
Phase III: generating models satisfying criteria *not* based on coverage.

In our work to date we have constructed a tool-chain which, when given a UML class diagram and a set of UML object diagrams, will calculate the three coverage measures described above [**?**]. The tool chain uses the USE tool as a parser and validator for the class and object diagrams, and uses the Eclipse Modeling Framework (EMF) to represent these as instances of the UML metamodel. To represent the output we have built a coverage metamodel in EMF which is essentially an extension of an existing metrics metamodel [**?**]. Finally, we have written a transformation to calculate the coverage measures using ATL (ATLAS Transformation Language).

In order to complete the first phase we intend to extend the coverage measures for class diagrams to deal with the associated OCL constraints. We intend to use (OCL) decision coverage as our initial measure, and to extend our tool chain to implement this coverage.

In the next phase, we hope to generate valid models that satisfy coverage criteria for at least one programming language metamodel. We are currently studying Alloy's approach as a model of exploiting third-party SAT solvers to control model generation.

At the final phase of our research we hope to expand this approach to other language-based criteria. To do this, we intend to use OCL-based queries across

the metamodel to specify the kind of model to be generated. Ideally, this would allow a user to specify the kind of programs that would be generated in the test suite for a given language metamodel.