Automated Test Input Generation for Software that Consumes ORM Models

Matthew J. McGill[1], R. E. Kurt Stirewalt[2], and Laura K. Dillon[1]

[1] Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48223
{mmcgill,ldillon@cse.msu.edu}
[2] LogicBlox, Inc.
kurt.stirewalt@logicblox.com

Abstract. Software tools that analyze and generate code from ORM conceptual schemas are highly susceptible to feature interaction bugs. When testing such tools, test suites are needed that cover many combinations of features, including combinations that rarely occur in practice. Manually creating such a test suite is extremely labor-intensive, and the tester may fail to cover feasible feature combinations that are counterintuitive or that rarely occur. This paper describes ATIG, a prototype tool for automatically generating test suites that cover diverse combinations of ORM features. ATIG makes use of combinatorial testing to optimize coverage of select feature combinations within constraints imposed by the need to keep the sizes of test suites manageable. We have applied ATIG to generate test inputs for an industrial strength ORM-to-Datalog code generator. Initial results suggest that it is useful for finding feature interaction errors in tools that operate on ORM models.

1 Introduction

Increasingly, tools are being developed to analyze and generate code from conceptual schemas specified in ORM 2.0. Examples include the professional edition of the Natural ORM Architect (NORMA) [1] and a tool we developed called *VisualBlox*, which analyzes ORM models to generate schemas in Datalog$^{\text{LB}}$ [2].[3] A key concern in the development of such a tool is the cost-effective design of a test suite that adequately tests the tool. In this context, a test suite is a large corpus of ORM models. This paper reports on a tool, called ATIG,[4] that automatically generates ORM test models covering diverse combinations of ORM features to use for testing purposes.

Automating generation of ORM test inputs presents several difficulties. For one, it can be difficult to ensure that the generated inputs are valid ORM. For

[3] The LogicBlox technology uses Datalog$^{\text{LB}}$ to program and query databases built atop the LB platform. Details on this language and the platform are beyond the scope of this paper.

[4] ATIG stands for Automatic Test Input Generator.

example, a generated model containing a fact type with no internal uniqueness constraint is not a valid ORM model. ATIG makes use of a specification of the ORM abstract syntax and semantics to avoid generating invalid test models.[5] A second difficulty for automating generation of ORM input models is producing reasonable-sized test suites whose inputs combine a wide range of ORM features in a variety of different ways. ATIG employs *combinatorial testing* [5] to achieve this goal.

ATIG takes as input a *feature specification*, which specifies the ORM features that the generated test suite should exercise, and a *test specification.* A feature specification encodes a subset of the ORM metamodel in a manner that facilitates automated analysis. The tool then processes these inputs to automatically produce a suite of ORM models that exercise the features described in the feature specification in a variety of combinations, using the test specification to drive the generation process. For convenience and due to space limits, we depict feature specifications as ORM metamodels expressed in ORM, even though they are currently written in Alloy.[6]

ATIG exploits two tools—the Alloy Analyzer, which automates the instantiation of relational logic specifications [4], and `jenny` [6], which automates the generation of combinatorial test plans. Because Alloy provides a host of logical and relational operators, representing ORM feature specifications in Alloy is straightforward. `Jenny` populates a candidate test plan while avoiding an explicitly specified set of *infeasible configurations.* A problem with using `jenny` for generating test plans is the need to know and explicitly enumerate infeasible configurations. We use Alloy's ability to find an instance of a specification, indicating a configuration is feasible, to iteratively refine candidate test plans, feeding configurations that Alloy cannot instantiate back to `jenny`. The process iteratively determines likely infeasible configurations until `jenny` is able to produce a test plan that can be fully instantiated. While this process may cause some possible feature combinations to be omitted, our experience to date indicates that the test suites it generates cover diverse combinations of ORM features and are effective in finding subtle interaction errors.

2 Background and Related Work

This section provides background on testing techniques and analysis tools used by our tool. In particular, we describe two complimentary testing techniques, category-partition testing (Sec. 2.1) and combinatorial testing (Sec. 2.2), aspects

[5] Determining the satisfiability of an arbitrary ORM model is NP-Hard [3]. However, the *small-scope hypothesis*, which argues that most errors can be found by exhaustively testing inputs within a small scope [4], suggests that the small test inputs generated by ATIG can find subtle interaction errors, and our experience to date supports this hypothesis.

[6] When NORMA supports first-class derivation rules, generating the Alloy specifications from feature specifications modeled in ORM will be straightforward.

Table 1. Example categories and choices for an array-sorting program

Category	Domain	Choices
`array_length`	Non-negative integers	0, 1, [2–10], >10000
`has_dups`	`bool`	`true`, `false`
`sort_order`	{`ascend`, `descend`}	`ascend`, `descend`

of which ATIG automates. We also provide a brief overview of the Alloy Analyzer (Sec. 2.3) and discuss related work (Sec. 2.4).

2.1 Category-Partition Method

The category-partition method (CPM) [7] is a general method for generating test inputs for software programs. A tester usually cannot exhaustively test a non-trivial software program, as the size of the program's input space is often too big for exhaustive testing to be practical. In CPM, the tester instead partitions the input space of a program into equivalence classes and selects at most a single input from each equivalence class for testing. A key property of this partition is that all test inputs from the same equivalence class should be sufficiently similar to be considered "equivalent" for testing purposes. This section briefly elaborate these ideas, eliding details of CPM not salient to our test generation method.

Before describing CPM, we need to introduce some terminology. To partition the input space, the tester identifies one or more *categories* and associated *choices* based on a specification of the valid program inputs. Intuitively, a category describes "a major property or characteristic" [7] of a program input by associating with each input a value from some *category domain*. For example, for an array-sorting program, a tester might define an `array_length` category, whose domain is the set of non-negative integers; a `has_dups` category, whose domain is `bool`, indicating whether the array contains any duplicate entries; and a `sort_order` category, whose domain is {`ascend`, `descend`}, as shown in Table 1.

In contrast, a choice for a category designates a subset of the category domain that contains values a tester considers equivalent for testing purposes. Choices for the same category are pair-wise disjoint, but they need not be exhaustive. For example, the choices for `array length` distinguish arrays of length 0 and 1, but identify all other small arrays (i.e., arrays containing from 2 to 10 elements) as equivalent and all very large arrays (i.e., arrays containing more than 10000 elements) as equivalent for testing purposes. Given a category and an associated choice, an input is said to *satisfy*, or equivalently, to *cover*, the choice if the value that the category associates with the input belongs to the choice. For instance, an input containing an array of length 5 satisfies (covers) the choice [2–10] for array length.

A *choice combination* is a set of choices for which each choice corresponds to a different category. A choice combination that additionally contains a choice for each category is also called a *test frame*. Given the three categories and corresponding choices in Table 1, for example, {1, `false`, `ascend`} is both a choice combination and a test frame, whereas any proper subset of it is just a choice combination. An input covers a choice combination if it covers each choice in the combination. Of course, categories are not necessary orthogonal. Thus, there might be no valid input satisfying certain choice combinations. For example, because arrays of length 1 cannot contain duplicates, the choice combination $\{1, \texttt{true}\}$ is infeasible.

In CPM, a tester provides a *test specification* in the form of definitions for a set of categories and corresponding choices, and a *test set* is generated from the test specification. Essentially, the set of all feasible test frames is viewed as defining a partition on the valid program inputs, where each equivalence class consists of the inputs satisfying one of the feasible test frames.[7] Ideally, therefore, a test set would cover all feasible test frames. In practice, however, it may not be practical to achieve this level of coverage. For one, determining feasibility is difficult (and can be undecidable). Moreover, the number of test frames grows exponentially with the number of categories, and so can simply be too large. Thus, the tester must typically select some subset of test frames to be covered. We refer to a subset of test frames to be covered by a test set as a *test plan.*

A test set is produced from a test plan by *instantiating* each of its test frames. Instantiation of a test frame involves searching for an input that satisfies all choices in the test frame. For example, to instantiate the test frame $\{[2\text{–}10], \texttt{false}, \texttt{ascend}\}$, an array of, say, length three might be populated with three arbitrary, but distinct, integer values, in which case the less-than operator for integers should be selected to use in comparing array elements. A key difficulty with instantiation is actually selecting valid inputs that simultaneously satisfy all the choices. Here again, inputs that are subject to complex syntactic and static semantic constraints (e.g., ORM models) compound this difficulty [3]. ATIG uses the Alloy Analyzer to automate instantiation of a test frame.

2.2 Combinatorial testing

As previously noted, a tester must typically narrow the set of test frames in CPM, both to eliminate infeasible test frames and to obtain a reasonable-sized test plan. We use combinatorial testing [5] for this purpose. Borrowing terminology from combinatorial testing, we say a test set achieves *t-way coverage* if it covers every feasible choice combination of size t. When t equals the number of categories, combinatorial testing and CPM produce equivalent test sets. But when t is small, t-way coverage can often be achieved using smaller test sets. For example, 2-way coverage of the categories in Table 1 can be achieved by a test set containing just 8 test inputs. As the number of categories increases, the reduction in test set size becomes increasingly significant, because a single test frame can cover more t-way choice combinations.

[7] If the choices for some category domains are non-exhaustive, then there will also be an equivalence class for all inputs that do not satisfy any test frame. We assume the tester does not care about covering these inputs and so ignore this equivalence class.

Tools, such as `jenny`, automate the generation of t-way test plans, provided the tester indicates infeasible choice combinations.[8] A key contribution of our work is to use the Alloy Analyzer in conjunction with `jenny` to automate the selection of a "good" test plan.

2.3 Alloy

Our current ATIG prototype works on a feature specification expressed in Alloy, a modeling language combining first order predicate logic with relational operators [4]. Additionally, it uses the Alloy Analyzer in instantiating a test frame or to classify a test frame as "likely infeasible."

More generally, the Alloy Analyzer generates instances up to some bounded size, or *scope*, of specifications expressed in Alloy. Instance generation is essentially a search problem and bounding the scope guarantees the search terminates. If an instance is found, then the Alloy specification is satisfiable. The Alloy Analyzer can display an instance graphically. It can also output a textual description of an instance in XML.

2.4 Related Work

Several existing methods for generating test inputs for code generators produce the inputs from a description of the source notation [8–10]. Those cited use UML class diagrams as the meta-notation, not ORM. Also, unlike our method, none attempts to systematically test combinations of source notation features.

Wang [8] and Brottier [9] use only information expressible in UML class diagrams, whereas Winkelmann *et al.* [10] augment UML class diagrams with limited OCL constraints to allow more precise source notation descriptions. The extra precision means that fewer invalid test inputs are generated. Neither ORM nor the OCL subset in [10] is strictly more expressive than the other. For example, OCL's implication operator has no natural representation in ORM, whereas ORM's transitivity constraints cannot be expressed in this OCL subset [10]. We expect, however, that a version of NORMA will have formal support for derivation rules that are at least as expressive as this OCL subset.

Other existing methods [11–13] require a formal specification of the code generator's mapping from source to target in addition to a description of the source notation. Formally specifying a code generator's behavior can provide many benefits. External constraints and rapidly changing requirements, however, can make formal specification of behavior impractical or impossible. During the development of VisualBlox, for example, user feedback motivated frequent changes to the code generator's requirements, and the complexities of the target language frustrated attempts at formality. In contrast, the description of the source notation remained fairly stable.

[8] The problem of generating minimal t-way test plans is NP-hard. Using a greedy algorithm, however, `jenny` quickly generates test plans that are near-minimal on average.

3 ATIG Inputs and Outputs

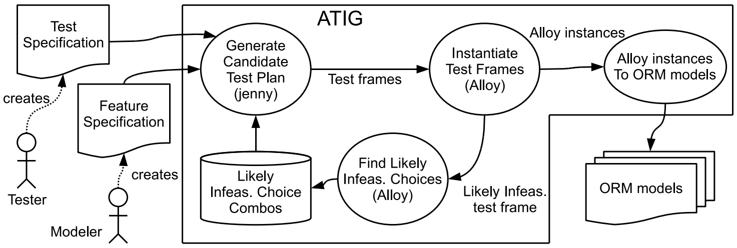

ATIG
Test Specification
creates
Tester
Feature Specification
creates
Modeler
Generate Candidate Test Plan (jenny)
Test frames
Instantiate Test Frames (Alloy)
Alloy instances
Alloy instances To ORM models
ORM models
Find Likely Infeas. Choices (Alloy)
Likely Infeas. test frame
Likely Infeas. Choice Combos

Fig. 1. Overview of ATIG

Figure 1 shows the inputs that ATIG uses and the outputs that it creates, as well as the main processes it performs and the data these processes produce and consume. We describe the inputs and outputs in this section, deferring discussion of the internals of ATIG to Section 4.

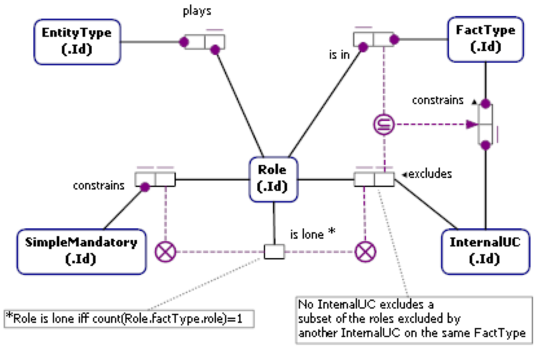

plays
EntityType (.Id)
is in
FactType (.Id)
constrains
constrains
Role (.Id)
excludes
is lone *
SimpleMandatory (.Id)
InternalUC (.Id)
*Role is lone iff count(Role.factType.role)=1
No InternalUC excludes a subset of the roles excluded by another InternalUC on the same FactType

Fig. 2. A simple ORM metamodel of ORM, for testing VisualBlox

ATIG takes as input a *feature specification*, which describes the ORM features of interest, and a *test specification*, which describes the categories and associated choices that a test set should exercise. Figure 2 illustrates the type of information expressed by a feature specification for a subset of the ORM features supported by VisualBlox. Because we do not have space to describe Alloy in this paper, we show the the feature specification as an ORM metamodel. Encoding this metamodel in Alloy is straightforward. The ORM metamodel represents ORM language elements by entity types a(e.g, `EntityType`, `Role`) nd represents potential relationships among language elements by fact types (e.g., `EntityType` `plays Role`). Constraints (e.g., simple mandatory constraints, exclusion constraints) capture some of the static semantics. Others are specified as derivation rules (e.g., rule labeled "*" attached to `Role is lone`).

For feasibility checking to be tractable, a feature specification can include only a relatively small subset of features and must encode static semantic constraints with an eye toward keeping the sizes of instances small. These considerations affect what to include in the feature specification and how to encode the static semantics. For example, an ORM metamodel would typically have a mandatory many-to-many fact type `InternalUC constrains Role`. However, encoding the compliment of this fact type—`InternalUC excludes Role`, a non-mandatory one-to-one fact type—produces smaller instances.

A test specification supplied to ATIG comprises a set of *cardinality categories* and associated choices. A cardinality category describes the number of times a particular ORM language element or language element relationship (described in the feature specification) appears in a test model. The domain of a cardinality category is the set of non-negative integers. By convention, a cardinality category is denoted `|element name|`, where `element name` stands for the ORM entity or predicate name from the feature specification (e.g., `|EntityType|`, `|Role is in FactType|`).

ATIG outputs a set of ORM test models in the format used by NORMA. It can also output these test models as diagrams, which facilitates checking that a test model is valid ORM.

4 ATIG: Test-set generation algorithm

ATIG takes as input a feature specification and a test specification, and generates a test set of ORM models using the iterative process depicted in Figure 1. This process employs `jenny` to iteratively generate *candidate test plans*, which achieve t-way coverage of the choices defined in the test specification. In addition to the space of choices, `jenny` takes as input a set of likely infeasible choice combinations to avoid when generating candidate test plans.[9] Initially empty, a set of likely infeasible choice combinations is gradually populated as a byproduct of attempts to instantiate the frames of a candidate test plan, as depicted in Figure 1.

ATIG uses the Alloy Analyzer to attempt to instantiate test frames. Based on the success or failure of these attempts, control passes along different arcs in Figure 1. Briefly, suppose F denotes an Alloy specification of a portion of the ORM metamodel and f denotes a test frame comprising a set of choices $\{c_1, c_2, \ldots, c_k\}$. ATIG instantiates f by:

1. translating f into an Alloy predicate P with k conjuncts, each of which encodes some $c_i \in f$ as a constraint on the cardinality of some signature or relation declared in F; and then

[9] `Jenny` treats each choice as an opaque identifier and thus may generate a host of infeasible choice combinations.

2. using the Alloy Analyzer to *run* P (within some finite scope) to generate an instance of F that satisfies all of the choices in f.

Step 1 is trivial because we use only cardinality categories and the names of these categories correspond to the names of signatures in F. Step 2 may have one of two outcomes: Either the Alloy Analyzer fails to find an instance within the specified scope, in which case we deem f is a likely infeasible frame, or it finds an instance, which ATIG then translates into an ORM model according to the schema of the XML representation used by NORMA. Rather than add a likely infeasible frame to the set of likely infeasible choice combinations, ATIG attempts to find a minimal subset of the frame that cannot be instantiated in the given scope.[10]

To summarize, ATIG generates a test plan comprising only test frames that are consistent with the feature specification, while attempting to optimize coverage of t-way choice combinations. It identifies likely infeasible choice combinations using an iterative process, incrementally building up a list of combinations for `jenny` to avoid until `jenny` produces a test plan that ATIG can instantiate (within some particular scope).

5 Discussion and Future Work

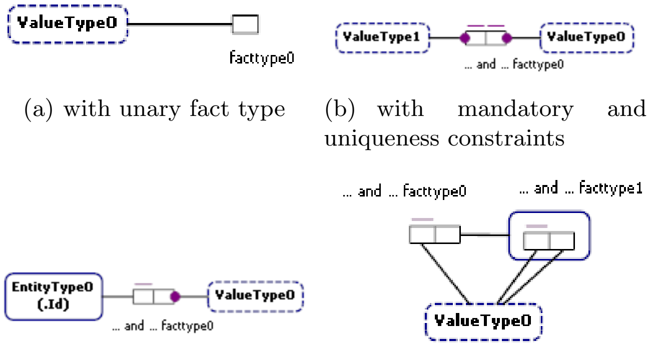
A small study with ATIG suggests the tool can generate moderate-sized test suites capable of exposing unexpected interaction bugs. In this study, we used a feature specification that extends the metamodel in Figure 2 with objectification and with integer and boolean value types. The extended metamodel contains 8 object types and 8 fact types, 2 of which are derived. For a test specification, we introduced a cardinality category for each non-derived fact type in the metamodel, and associated 5 choices—namely, 0, 1, 2, [3,5], and [6–8]—with 3 of the categories and 4 choices—namely, 1, 2, [3,5], [6-8]—with the other 3 categories.

We ran ATIG on these feature and test specifications to generate a 2-way test set of ORM models. The test specification produces a total of $8,000$ distinct test frames. ATIG generated a test plan containing just 37 test frames, classifying 51 choice combinations as likely infeasible in the process. It took just under 19 minutes (wall time) on a desktop computer containing an Intel Core 2 Duo 3 ghz processor and 2 gigabytes of RAM. The majority of this time is spent in checking choice combinations for feasibility when a test frame cannot be instantiated within the scope used.

The 37 models in the generated test set cover a diverse range of ORM feature combinations and exposed a number of previously unknown errors in the current version of VisualBlox. Figure 3 shows four ORM models from the generated test set. Collectively, the models in Figure 3 cover value types used in combination with every other ORM language element in the feature specification, namely entity types, simple mandatory and internal uniqueness constraints, fact types

[10] Providing smaller choice combinations will prevent `jenny` from using these combinations in any generated test plan, speeding convergence of the iterative process.

of different arities, and objectification. By covering many feature combinations not present in our manually produced test set, the generated test set uncovered multiple, previously unknown errors. Specifically, 15 of the 37 test models, each a valid ORM model, exercise ORM features in combinations that caused VisualBlox to produce Datalog code that was not accepted by the Datalog$^{\text{LB}}$ compiler. Thus, roughly 40% of the automatically generated test inputs were valid models that the VisualBlox development team had not considered supporting.



(a) with unary fact type

(b) with mandatory and uniqueness constraints

(c) with entity types

(d) with objectification

Fig. 3. ATIG-generated models combining value types with other ORM modeling features

Our preliminary results suggest several areas for future work. First, we plan studies to more formally assess the quality of test sets generated with ATIG. Our first study will include more ORM features in the feature specification, and compare the cost-effectiveness of 2-way, 3-way and higher, and manually constructed test sets. Later studies will evaluate various heuristics used in our current prototype. For example, how does using smaller scopes when instantiating test frames affect the size of the generated test plan, the diversity of the choice combinations covered by this plan, the errors exposed by a test set, and the time to generate test sets? Another question deserving more attention in light of the time spent finding minimal-sized likely-infeasible choice combinations is the cost-effectiveness of this step. Would equally diverse test sets be produced if we collected test frames instead of minimal choice combinations, and how would doing so affect the time for the iterative process to converge? In other future studies, we plan to evaluate whether ATIG can provide similar benefits if used to generate test sets for programs that consume other types of inputs with complex structural constraints (e.g., UML models).

To facilitate conducting such studies, we also plan to automate the translation of an ORM metamodel into an Alloy feature specification. Translating the graphical ORM language features to Alloy is straightforward. At present, textual constraints on derivation rules pose the primary obstacle to automating the translation to Alloy because there is no formal textual syntax for constraints in NORMA. We anticipate that a future version of NORMA, however, will support textual derivation rules for derived fact types and textual constraints.

Another promising area for future work is adding support to ATIG for parallel invocations of Alloy on separate processors or computers to take advantage of the embarrasingly parallel nature of instantiating test frames and searching for likely infeasible choice combinations.

References

1. Curland, M., Halpin, T.: Model driven development with NORMA. In: Proc. of the 40th Int'l. Conf. on Sys. Sci. (2007)
2. Zook, D., Pasalic, E., Sarna-Starosta, B.: Typed datalog. In: Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages. (2009)
3. Smaragdakis, Y., Csallner, C., Subramanian, R.: Scalable automatic test data generation from modeling diagrams. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ACM New York, NY, USA (2007) 4–13
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
5. Cohen, D., Dalal, S., Parelius, J., Patton, G., Bellcore, N.: The combinatorial design approach to automatic test generation. IEEE software **13**(5) (1996) 83–88
6. `http://burtleburtle.net/bob/math/jenny.html`: jenny (June 2006)
7. Ostrand, T., Balcer, M.: The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM **31**(6) (1988)
8. Wang, J., Kim, S., Carrington, D.: Automatic Generation of Test Models for Model Transformations. In: Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on. (2008) 432–440
9. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on. (2006) 85–94
10. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. Electronic Notes in Theoretical Computer Science **211** (2008) 159–170
11. Sturmer, I., Conrad, M., Doerr, H., Pepper, P.: Systematic Testing of Model-Based Code Generators. IEEE Transactions on Software Engineering **33**(9) (2007) 622–634
12. Baldan, P., Konig, B., Sturmer, I.: Generating test cases for code generators by unfolding graph transformation systems. Lecture notes in computer science (2004) 194–209
13. Lamari, M.: Towards an automated test generation for the verification of model transformations. In: Proceedings of the 2007 ACM symposium on Applied computing, ACM New York, NY, USA (2007) 998–1005