2010

This is to certify that the
thesis entitled

GENERATING COMBINATORIAL TEST SETS FOR MODEL TRANSFORMATIONS

presented by

Matthew J. McGill

has been accepted towards fulfillment
of the requirements for the

Master of Science	degree in	Computer Science

[illegible]

Major Professor's Signature

May 14, 2010

Date

GENERATING COMBINATORIAL TEST SETS FOR MODEL TRANSFORMATIONS

By

Matthew J. McGill

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science

2010

ABSTRACT

GENERATING COMBINATORIAL TEST SETS FOR MODEL TRANSFORMATIONS

By

Matthew J. McGill

Model transformations are particularly susceptible to feature interaction errors. To expose feature interaction errors during testing, a developer must construct test inputs that systematically cover modeling language features in combinations. Manually constructing such a test set is labor-intensive, and the developer may fail to cover rarely occurring or counter-intuitive feature combinations. This thesis describes ATIG, a prototype tool for automatically generating sets of models to test model transformations. ATIG uses combinatorial testing techniques to generate test sets of manageable size that systematically cover the language features of a model transformation's source notation. To validate ATIG, I used it in a case study to generate a test suite for an industrial strength code generator. The results suggest that ATIG is useful for finding subtle feature interaction errors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Software developers most commonly verify a program by *testing* the program on a variety of inputs. Testing involves executing a program one or more times for each input in a *test set*, and comparing in each case the *actual output* of the program with the *expected output* for that input. If the actual and expected outputs disagree, then the program or the specification of the expected input (or both) contains an error, and we say that the test *exposes* the error. Testing is an empirical process, and cannot generally prove the correctness of a program in a mathematical sense. However, because testing is much less expensive than formal verification, and because testing demonstrates that a program is correct for at least some inputs, it is overwhelmingly the most common program verification approach.

Some test sets are better than others at exposing errors, and researchers have developed many techniques for selecting test sets [16, 21, 9, 13, 3, 5]. The Category-Partition Method [21] (CPM) is a *black-box* method, wherein test inputs are selected from a partition of the input space that is derived from a functional specification of program behavior. Another method, t-way Combinatorial Testing [5] (hereafter t-way testing), applies combinatorial design to cover many combinations of input parameter values with as few tests as possible. These methods have achieved popularity in the software development industry, in part because they can be automated: associated tools can automatically generate descriptions of the inputs to include in a test set. Additionally, the methods are complementary: t-way testing can be used in conjunction with the CPM to produce smaller test sets that are still regarded likely to expose errors when they exist.

These methods, however, have proven difficult to apply when the inputs to a program must satisfy complex structural constraints. Such is the case for an important class of programs called *operational model transformations.* Operational model transformations (hereafter simply model transformations) translate source software models into target models or implementation-level code [11]. Model transformations play a key role in *Model-Driven Engineering* (MDE), a designation for software development approaches that use software models to bridge the gap between software requirements and implementations [11]. Model transformations have many uses in MDE, including automated refinement and refactoring of models, and automated code generation. They must be thoroughly tested so that errors are not introduced into transformed models. We would like to use the CPM and t-way testing to generate models for model transformations, but in this context the existing tools for the CPM and t-way testing are difficult to apply. Most existing tools have insufficient support for expressing or handling constraints [5], such as those embodied by the static semantics of a modeling language. When the tools are used to generate descriptions of test sets comprising models, the descriptions are frequently inconsistent with the static semantics of the modeling language and therefore not useful for testing.

This thesis explores the possibility of compensating for weak constraint support in existing tools by integrating them with additional technologies. Specifically, I evaluate the following three hypotheses:

1. Existing tools for the CPM and t-way testing can be extended to generate test sets of models that respect complex static semantic constraints.

2. The generation of test sets for model transformations can be fully automated.

3. The resulting test sets are useful for finding errors in real-world programs.

Chapter 2 provides necessary background information, including the main technologies upon which my research builds. Chapter 3 introduces the Automatic Test Input Generator (ATIG), the tool that I developed to automate the generation of t-way test sets, according to the CPM, that respect complex constraints. Chapter 4 describes a case study in which I used ATIG to automatically generate a test set for an industrial code generator. Chapter 5 discusses prior work on generating test inputs, examining how each work relates to my approach. Finally, Chapter 6 presents my conclusions, and includes a discussion of planned future work in the area of automated test input generation.

Chapter 2

Background

This chapter describes the testing methods that motivate ATIG, and the tools that ATIG employs to generate test inputs:

- In Sections 2.1 and 2.2 I discuss the CPM and *t*-way combinatorial testing respectively. ATIG was developed to support the application of these methods to the testing of model transformations.
- Section 2.3 introduces jenny, an existing tool that ATIG relies on to generates *t*-way combinatorial test sets.
- Section 2.4 describes the ORM modeling notation, which I employ to describe the static semantics of modeling languages;
- Finally, Section 2.5 describes the Alloy Analyzer, which I use to generate test sets that respect the static semantics.

2.1 Category-Partition Method

A tester that constructs a set of test inputs in an ad-hoc manner can easily overlook important important cases that should be tested. The *Category-Partition Method* [21] (CPM) is a *systematic* method of deriving a set of test inputs from a program's specification that is intended to overcome this problem.

The CPM is a six-step process.

1. The tester analyzes a program's functional specification to identify functional units of the program that can be tested individually. For each functional unit, the developer analyzes the specification to find characteristics of the unit's parameters or environment that affect its behavior in a way that should be tested. Each identified characteristic is referred to as a *category*.

2. Each category is partitioned into *choices*, where each choice is a significant case within a category.

3. The tester determines constraints among choices from different categories. It is unusual for all identified categories to be completely orthogonal. The tester must identify which combinations of choices do not describe valid parameter values or environmental conditions.

4. The tester writes a *test specification* containing the category, choice, and constraint definitions, and supplies it to a program that generates a set of *test frames* that cover all choice combinations not excluded by a constraint. A test frame is a set of choices, one from each category, that defines a single

equivalence class in a partition of the program's input space.

5. The tester examines the set of test frames in case the test specification needs revision. The absence of a key test scenario, an illegal choice combination, or an unreasonably large number of frames might cause the tester to amend the test specification and generate a new set of test frames.

6. Having ensured that the set of test frames is acceptable, the tester converts each frame into a test case.

Steps 3-5 are iterated until a satisfactory set of test frames has been produced.

To summarize, a tester analyzes the specification to identify important characteristics of the program's input parameters and environment. The tester then uses these characteristics to derive a partition on the input space of the program, and selects a single test input from each equivalence class.

A simple example clarifies the CPM steps. We are given a specification for a web application login page that prompts the user for a user name and password. According to the specification, a user name consists of up to 32 alphanumeric characters; a password must be between 6 and 16 characters in length. On successful login, the user is forwarded to a home page. When the user name is unrecognized or the password is incorrect, the user remains at the login page, and an error is displayed. Three unsuccessful attempts locks the user out of the application for a period of time. We can use the CPM to derive, from the specification, an effective set of inputs to test this login screen.

In step one, we first identify functional units, or modules, that may be tested individually. We next extract categories from the functional specification. In our simple example, the specification clearly describes a single functional unit. Several important characteristics of the inputs are apparent:

1. the length of the entered user name,
2. the kinds of characters present in the user name (i.e. alphanumeric or not), and
3. the length of the entered password.

Environmental characteristics include[1]:

1. whether the entered user name is recognized by the system,
2. whether the password is correct, and
3. the number of prior unsuccessful login attempts.

In step 2, we select choices for each category. Table 2.1 shows possible choices for the categories in our example. Notice that the choices pay particular attention to boundary conditions. In general, choices represent important cases to test; most are taken from the specification, but we may also rely on experience or special knowledge of the implementation (for example, including unrealistically large inputs to check for buffer overflows).

In step 3 we determine constraints among the choices. In our example the categories are completely orthogonal, and there are no constraints to consider.

[1]One might argue whether the correctness of the password is a characteristic of the input or the environment. Either answer is defensible, and the distinction is not important when generating test frames.

Table 2.1: Example test specification for a web application login page

Category	**Choices**
User name length	0; 1-31; 32; 33; 34-1499; 1500
User name characters	Only alphanumeric; Alphanumeric with spaces; Non-printable characters
Password length	0; 1-5; 6; 7-15; 16; 17; 18-1499; 1500
Password characters	Only alphanumeric; Alphanumeric with spaces; Non-printable characters
Prior failed login attempts	0; 1; 2; 3

In step 4, we would represent the categories, choices, and constraints as a test specification in the input language of a generator tool (such as jenny, described in Section 2.3). We would then use the generator tool to produce a set of test frames containing all choice combinations not excluded by the constraints. An example of a test frame that might be generated from the choices in Table 2.1 is the tuple (1-31, "Only alphanumeric", 0, "Alphanumeric with spaces", 3).

Assume the set of generated test frames was acceptable (step 5). In step 6, we would convert each frame into a complete test case with inputs and expected outputs. In our example, we might turn each test frame into a test script, with instructions for how to manually prepare the environment, enter appropriate inputs into the login form, and verify the results. Alternatively, we might convert each test frame into a single test case for an automated GUI testing tool such as Selenium [12].

In theory, we can apply the CPM to any program with an associated specification. In practice, the CPM is difficult to use on model transformations because the constraints on choice combinations are difficult to identify. The difficulty lies in accounting for the static semantics of the transformation's source notation. Identifying which choice combinations are incompatible with the static semantics of a modeling language can be extremely challenging when the static semantics are complex. [2] Ideally, a CPM tool for model transformations would *automatically* determine the choice combination constraints by analyzing a formal representation of the source notation's static semantics.

2.2 Combinatorial testing

t-way combinatorial testing [4] is a testing method for covering many combinations of input parameters with as few test cases as possible using *t-way combinatorial test sets*. A t-way combinatorial test set is a test set that covers all combinations of parameter values for every set of t parameters. The method relies on the intuition that only the values of a small subset of input parameters determine whether a given error will be exposed by a test case. If this intuition is true, most errors can be exposed by testing all combinations of carefully selected candidate values for each set of t input parameters. The benefit of using small values of t is a dramatic reduction in the number of necessary test cases. When t is smaller than the total number of parameters, each test case can cover multiple size-t combinations. D. Cohen *et al*[3] [4] show that the size of an optimal t-way combinatorial test set grows logarithmically in the number of parameters for fixed t.

t-way combinatorial testing and the CPM are complementary methods. Whereas the CPM focuses on identifying the characteristics of a program to be considered

[2]Consider that the static semantics of UML 2.x are scattered throughout a 700-page specification [30].

[3]Here and in the remainder, we distinguish between David Cohen and Myra Cohen.

during testing, t-way combinatorial testing focuses on efficiently covering combinations of input values. The CPM can produce an impractical number of test frames to instantiate; the number of frames grows exponentially in the number of categories. In the standard CPM, the tester reduces the number of test frames by adjusting the test specification, either by removing choices and categories or by adding additional constraints to restrict the allowed choice combinations. Alternatively, a tester may reduce the number of frames by using a tool to generate a t-way combinatorial test set for some t less than the total number of categories. The reduction in test set size can be dramatic. There are 59,049 possible test frames for 10 categories of 3 choices each. In contrast, there is a 3-way test, i.e. a test set covering every combination of choices for each group of three categories that contains just 65 test cases.

2.3 Combinatorial testing with jenny

ATIG uses a tool called jenny [28] to generate t-way combinatorial test sets. The inputs to jenny are:

1. a value of t,
2. a sequence of numbers denoting the number of choices in each category, and
3. an optional set of choice combinations to exclude from the generated test set.

jenny uses a greedy algorithm to produce a small t-way test set from these inputs. Categories and choices are represented generically in the generated test set. Categories are assigned numbers increasing from 1, and each choice in a category is assigned a single letter increasing from ‘a’. To use the test specification, a tester associates each category number and choice letter with a specific category and choice from the test specification.

2.4 Object-Role Modeling

A formal description of a modeling language and its static semantics is a prerequisite for generating valid models in that language. I use the Object-Role Modeling [15] (ORM) language to describe a modeling language's abstract syntax and static semantics. ORM is a graphical fact-oriented language for conceptual modeling: an ORM model represents the world in terms of objects (things) and the *facts* that are known about how they relate. Unlike many modeling approaches, such as Entity-Relationship (ER) modeling and the UML, which force modelers to divide facts into relationships (relating objects) and attributes (relating objects with values), ORM makes no attribute-relationship distinction. The decision to represent a kind of fact as an attribute or a relationship is not always clear; when it is not, the necessity of choosing becomes an obstacle. By avoiding the attribute-relationship distinction, ORM simplifies the task of creating high-level conceptual models to describe almost any domain.

Figure 2.1 shows a project management conceptual model for a software development company. The figure illustrates all of the ORM features that we discuss in this section.

The *things* described by an ORM model are represented with *object types*. An

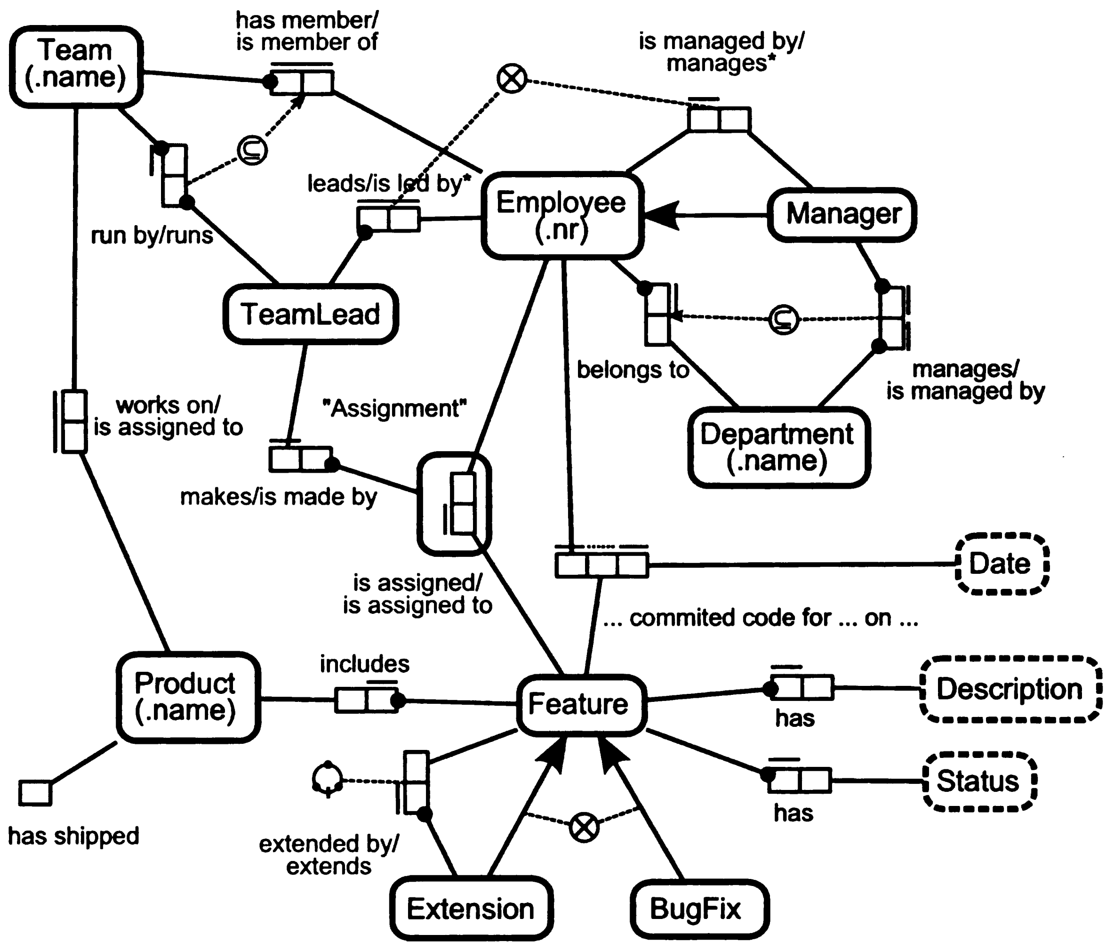


Figure 2.1: Example ORM product management data model
ORM model of product management in a hypothetical software company

object type denotes a set of objects, and is drawn as a rounded rectangle containing a name. There are two kinds of object type: *entity type* and *value type*. Loosely speaking, an entity type is a set of instances (or entities) that are distinct from the labels we use to identify them. The person Jane is an entity in the entity type Manager. The label "Jane" is distinct from, and uniquely identifies, a physical object. In Figure 2.1, Team(.name), Project(.name) and Employee(.nr) are entity types. The abbreviations in parentheses indicate an entity type's *reference scheme*: in our example, we uniquely identify teams and projects by their names, and employees by their employee number. When referring to an entity type, we often elide its reference scheme for brevity. In contrast, the instances of a value type are just values[4]. A value type is drawn as a rounded rectangle with a dashed border to distinguish it from an entity type. In Figure 2.1, Date, Description, Status are value types.

Table 2.2: Sample population of an ORM model
Sample population for a fact type and two entity types from Figure 2.1

Team(.name)
UI
Optimization
QA

Team works on Product	
UI	Word Processor
UI	Spreadsheet
Optimization	Spreadsheet

Product(.name)
Word Processor
Spreadsheet
Printer driver

Fact types represent relationships among object types. A fact type contains one or more *roles*, each played by an object type, and is drawn with adjacent boxes (one for each role) connected to the role players. A fact type may have any arity (number of roles), though binary fact types are most common. A fact type has one or more *readings*; a reading is a natural-language statement with place-holders for each role player. A reading is written alongside its fact type; a '/' separates alternative readings. For example Team works on Product is a fact type that can also be read Product is assigned to Team. As the reading suggests, facts in this fact type record which teams work on which products.

The population of a fact type can be visualized as a table with a column for each role. Table 2.2 shows a population for Team works on Product along with the populations of the role-playing entity types. We interpret ORM models according

[4]The entity type/value type distinction is not always crystal clear; in most cases, for example, a calendar date is considered a value type. However, in certain contexts it may be appropriate to model a date as an entity type identified by, for example, a unique combination of month, day and year. The subtleties of the entity type/value type distinction are not important for understanding this thesis.

to the *Closed-World Assumption* (CWA), the assumption that all relevant facts are known and reflected in a model's population. According to the CWA, we can deduce from Table 2.2 that team "QA" does not work on any product and that product "Printer driver" is not assigned to any team.

```
Some TeamLead leads some Employee
iff
    that TeamLead runs some Team and
    that Employee is member of that Team
Some Manager manages some Employee
iff
    that Manager manages some Department and
    that Employee belongs to that Department
```

Figure 2.2: Derivation rules for derived fact types in Figure 2.1.

Most fact types are *asserted* fact types. The population of an asserted fact type cannot be deduced *a priori* from any information in the model; its population of facts must be given. However, the populations of some fact types may be derived from the populations of other fact types in the model; we call these *derived* fact types. A derived fact type is marked with an asterisk ("*"). Figure 2.1 includes two derived fact types: TeamLead leads Employee and Employee is managed by Manager. The contents of these fact types can be derived according to the *derivation rules* given in Figure 2.2. Derivation rules can be included directly in the model.

ORM provides a rich set of constraints for conceptual modeling. The most often used constraints are simple mandatory and internal uniqueness constraints. A simple mandatory constraint applies to a single role, and indicates that the role must be played at least once by every instance of the role-playing object type. The constraint is drawn as a solid dot where a role connector line meets a role's box.

The simple mandatory constraint on the Employee role of Employee belongs to Department states that every employee belongs to at least one department. Internal uniqueness constraints apply to one or more roles in a fact type, and indicate that each combination of role players can occur at most one time in the fact type population. Internal uniqueness constraints are drawn as lines above a fact type's roles. In the example, the internal uniqueness constraint on Product includes Feature says that a feature is included in at most one product. The role played by Feature also has a simple mandatory constraint. The two constraints together express that each feature is included in exactly one product. Employee committed code for Feature on Date has an internal uniqueness constraint spanning two of its three roles (the dotted line over Feature's role indicates that the role is not included). This constraint states that for a given employee and a given date, the employee committed code for at most one feature on that date.

ORM also supports subtyping of object types. A subtype relationship is indicated by an arrow pointing from the subtype to the supertype. In our example, TeamLead and Manager are subtypes of Employee. ORM's subtyping feature is very flexible; a subtype relationship by itself states only that the instances of the subtype are a subset of the instances of the supertype. Subtypes are neither disjoint nor exhaustive by default. In our example, an employee might not be a team lead or a manager, might be one but not the other, or might be both a team lead and a manager.

We have seen fact types that represent facts about object types; occasionally modelers wish to represent facts about fact types themselves. A modeler can represent facts about a fact type by *objectifying* the fact type. An objectified fact type has an associated entity type, and each fact in the fact type is associated one-to-one with an entity in that entity type. An objectified fact type can thus play roles in fact types like any other entity type. An objectified fact type is drawn inside a rounded rectangle, like an entity type. In Figure 2.1, Employee is assigned Feature is objectified as Assignment, allowing us to express that a TeamLead makes (an) Assignment.

A *value constraint* restricts the set of values in a value type, and is drawn as comma-delimited list or range next to the value type. In our example, a value constraint on Status lists the possible status values as "unstarted", "implemented", and "tested".

A *frequency constraint* indicates how many times a role player must participate in a given fact type. It is shown as a number or inequality attached to a role with a dashed line. A frequency constraint may express a minimum number of times, a maximum number, or both. The frequency constraint on the Employee role of Team has member Employee states that each team that plays a role in the fact type has at least three team members. Frequency constraints apply to the objects that *play a role* in a fact type, not to all objects in the object type that plays a role. So, the frequency constraint on Team has member Employee does not by itself require *all* teams to have three employees; only those that have *some* employee must have at least three. However, because a simple mandatory constraint requires that all teams have *some* employee, the net effect in this case is to require that teams have three or more members.

Exclusion constraints apply to sequences of roles to indicate that no combination of role-playing objects occurs in more than one role sequence. Exclusion constraints are very powerful because they can express constraints across multiple fact types. They are drawn as Xs inscribed in a circle, with dashed lines connecting the constrained role sequences. In Figure 2.1, an exclusion constraint covers the roles in TeamLead leads Employee and Employee is managed by Manager. The constraint states that no team lead both leads and is managed by the same employee; that is, no team lead may be in charge of their manager. Exclusion constraints may also be applied to subtype relations, as with Extension and BugFix. In this context, the constraint states that the two subtypes are disjoint.

Disjunctive mandatory constraints generalize simple mandatory constraints. Disjunctive mandatory constraints apply to a set of roles played by the same object type[5], and state that each instance of the object type must play in *at least one* of the roles. A simple mandatory constraint may be thought of as a disjunctive mandatory constraint that covers only one role. Like exclusion constraints, disjunctive mandatories can be applied to subtype relations, where they indicate that each instance of the supertype must also be an instance of at least one subtype. Disjunctive mandatory constraints are drawn as a solid dot contained in a circle, connected by dashed lines to the covered roles. When a disjunctive mandatory constraint and exclusion constraint cover the same roles, they are often superimposed on one another to form a single *exclusive-or* constraint.

Subset constraints are used to indicate that the population of one role sequence is

[5]In fact, disjunctive mandatory constraints can apply to roles played by different types, provided the types share a common ancestor type.

a subset of the population of another role sequence. Subset constraints are drawn as a dashed arrow from the subset role sequence to the superset role sequence, passing through a circled subset symbol. Our example contains two subset constraints. One of these constraints states that every manager that manages a department also belongs to that department.

Finally, *ring constraints* apply to pairs of roles played by object types with a common ancestor type. In Figure 2.1, a ring constraint is applied to "Feature is extended by Extension". Ring constraints are so-called because the path from one role, through its role-playing object type, to the other role, forms a ring. There are several kinds of ring constraints; the ring constraint in our example is an *acyclic* ring constraint. It means the "Feature is extended by BugFix" relationship contains no cycles; informally, no chain of feature extensions can loop back on itself.

2.5 Alloy and Alloy Analyzer

Alloy Analyzer [18] represents a compromise in the use of formal methods versus testing to validate software. Ideally, software developers would rely on formal methods to prove conclusively that their software is correct and free of errors. Unfortunately, formal methods of proof have so far been too difficult and expensive to apply to all but the most safety-critical software. In practice, testing is most often used to build confidence in the correctness of a software system because it is easier and less expensive. However, it is commonly observed that testing can at best demonstrate the presence of errors, never their absence; thus testing provides less assurance than formal proof. Alloy Analyzer aims to provide some benefits of each approach by enabling automated, rigorous analysis to prove or disprove properties of formal models *within a specified finite scope.* As with formal methods of proof, a user can formally specify and verify properties against a formal model. However, the properties are only assured for instances less than a specified *scope*, or size, and the "proofs" are arrived at via an exhaustive search for a counterexample, which gives the tool the flavor of exhaustive testing. Alloy Analyzer also allows validation by examination of individual instances of a model, which the tool can generate automatically.

Alloy Analyzer analyzes models expressed in the Alloy language. Alloy comprises first-order logic (variables, boolean operators, and quantification) with the addition of relational operators and transitive closure. Alloy's semantics are based entirely on relations; sets are considered unary relations, and elements are considered singleton sets. An Alloy model contains *signatures*, relations, functions, predicates, and facts. A signature introduces a type, or named set, of atoms. Alloy supports sub-typing of signatures. Relations among signatures can be of arity two or higher. Functions are parameterized expressions that evaluate to a value (i.e. an atom, set or relation); they are used to remove repetition and increase comprehension in models. Predicates are functions that return true or false, and are useful for exploring a model's instances; Alloy Analyzer can generate instances for which a given predicate evaluates true, or show than no such instances exist within a given scope. Finally, facts are expressions (possibly referencing functions or predicates) that are defined to be true of any instance of the model.

Alloy Analyzer provides several analysis capabilities. Most simply, it can iterate through a model's instances, allowing the modeler to validate a model by ensuring that successive instances agree with the modeler's intentions. When a model has no instances within the chosen scope, the tool reports that the model may be inconsistent. The tool can also search for instances that satisfy predicates, enabling the modeler to examine instances having certain specific characteristics.

In addition to finding instances, the tool can be used to verify *assertions.* An assertion is a statement that the modeler believes to be entailed by the model's facts. Alloy Analyzer can show that the assertion holds within a specified scope, or provide a counter-example if it does not. Assertions are useful for checking that the modeler's assumptions about a model are correct.

The analyses performed by Alloy Analyzer are ultimately brute-force searches, and are subject to a *combinatorial explosion* of the search space; the search space increases exponentially with the size of the model and the chosen scope. Alloy Analyzer reduces instance generation and analysis tasks to instances of the *3-satisfiability problem*, or 3SAT, for which no efficient algorithms are known to exist. Given a finite boolean expression of conjuncts of the form $(p_i \vee p_j \vee p_k)$, where each p_i is either a variable or its negation, the 3SAT problem is to find an assignment of the variables that makes the expression true, or show that none exists. The tool relies on highly optimized 3SAT solvers that are able to handle 3SAT problems with tens or hundreds of thousands of variables. Due to combinatorial explosion, use of Alloy Analyzer is restricted to small models and scopes. Alloy Analyzer's usefulness as a validation tool despite these scalability problems is based on the *small scope hypothesis.* The small scope hypothesis holds that most errors in a program or model can be exhibited by means of small test cases or examples [18].

Chapter 3

Generating Combinatorial Test Sets of Models

The Automated Test Input Generator (ATIG) integrates existing tools to enable automatic generation of t-way test sets for model transformations. ATIG generates test inputs that conform to a tester-supplied *feature specification*[1], which is a machine-readable metamodel that describes the abstract syntax and static semantics of a model transformation's source notation. ATIG leverages the jenny tool to generate t-way test plans, and the Alloy Analyzer to search for populations of the feature specification that conform to each test frame in a test plan. Existing combinatorial testing tools require the user to manually identify infeasible choice combinations. An infeasible choice combination is a choice combination that only describes invalid test inputs. ATIG represents a significant contribution because it automatically infers *infeasible choice combinations* that arise from constraints in the

[1]i.e. a specification of the relevant language *features* from the source notation

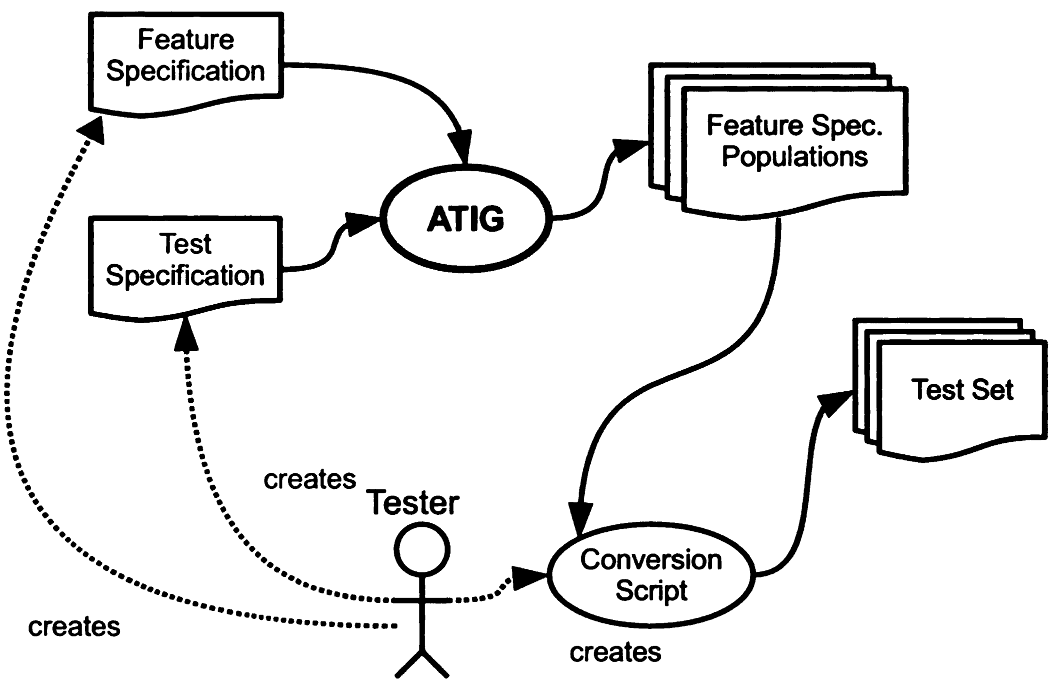


Figure 3.1: Overview of the ATIG tool

feature specification.

3.1 ATIG Inputs and Outputs

Figure 3.1 provides an external view of the ATIG tool. To generate test inputs for model transformations using ATIG, a tester first creates a feature specification to describe the syntax and static semantics of the source notation. The tester next creates a test specification that describes how to partition the input space for testing. The tester runs ATIG to generate a set of *populations* of the feature specification; each population represents a valid test input. Finally, the tester converts each population into a test input. The tester typically automates the conversion process with a script.

Separating the feature specification and the test specification allows the feature specification to be reused. A tester has the option of creating multiple test specifications that each focus on separate parts of one feature specification.

3.1.1 Feature Specification

The feature specification, as previously noted, is a metamodel. Metamodels are commonly expressed in the literature using UML class diagram notation, or a similar attribute-based notation. I chose ORM to express the feature specification because, in my view, ORM is an excellent language for metamodeling. First, ORM has a clean graphical syntax that, when coupled with tool support to manage multiple diagrams relating to a single model, can be used to model very complex structures in an intuitive, readable fashion. ORM is frequently used by business analysts as a tool to communicate with customers, and was designed specifically to be understandable by nontechnical users. As such, ORM has a shallow learning curve that will represent a low hurdle for testers looking to make use of ATIG. Second, ORM has a formal semantics grounded on set theory and first-order logic. A formal semantics ensures that each ORM model can be interpreted unambiguously, and serves as a foundation for applying tools such as the Alloy Analyzer to reason about and generate instances of ORM models. Finally, ORM has robust tool support in the form of NORMA [7], an extension to Microsoft Visual Studio for creating ORM models. ATIG consumes ORM models stored in the file format used by NORMA. To summarize, ATIG relies on ORM as a metamodeling notation because ORM has a clean, understandable

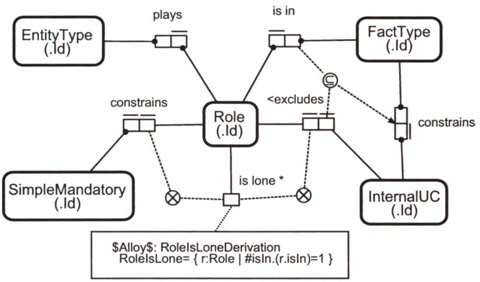


Figure 3.2: An example feature specification for ORM
An example feature specification for ORM models, expressed in ORM

graphical syntax as well as a formal semantics that enables instance generation.

Figure 3.2 shows a simple example of a feature specification that might be used to generate simple ORM models[2] A population of this model represents a (different) ORM model, for example the ORM product management model in Figure 2.1. Object types represent ORM modeling concepts; the object types Role, FactType and EntityType, for example, represent the populations of roles, fact types, and entity types in an ORM model. The fact types in the model represent the ways that ORM modeling elements can be composed together. InternalUC constrains FactType shows that an internal uniqueness constraint applies to a unique fact type. The constraints in the metamodel express ORM's static semantics. The exclusion constraint on the roles played by Role in Role is lone and InternalUC excludes Role, for example,

[2]Figure 3.2 is both an ORM model and a description of the ORM modeling languages; thus, it is a *reflexive metamodel* [22]. A reflexive metamodel can be confusing, but we use one as an example because the case study uses a more complex reflexive metamodel of ORM to generate test inputs.

captures the static semantic constraint that roles in unary fact types cannot have internal uniqueness constraints.

Most features and static semantic constraints for a modeling language can be expressed graphically in ORM. However, some constraints and derivation rules may not be expressible using ORM's graphical syntax. For this reason, ATIG supports the use of *textual constraints*. We express textual constraints in Alloy, and embed them in a feature specification as notes. ATIG processes these notes along with the rest of the feature specification when generating test sets[3].

Figure 3.2 shows a derivation rule for the unary fact type Role is lone. The **$Alloy$** symbol informs ATIG to process the note as a constraint or derivation rule, depending on the context[4]. The rule states that Role is lone is the set of roles belonging to fact types that contain a single role.

3.1.2 Test specification

Whereas the feature specification defines a "space" of input models, the test specification dictates how that space is partitioned for testing. In addition, the test specification includes several configuration options. Figure 3.3 shows a possible test specification for the feature specification presented above in Figure 3.2. The **T** parameter (line 1) is the value to use for t when generating a t-way combinatorial test

[3]At this time the NORMA modeling tool does not support a formal textual notation, but we expect this support to be available soon (see [14] for a brief overview of the planned notation). It will be straightforward to extend ATIG to support textual constraints in the language that is ultimately adopted within NORMA.

[4]The names in the derivation rule must conform to ATIG's rules for mapping ORM models into Alloy code. In this example, RoleIsLone refers to the fact type Role is lone, Role to the entity type of the same name, and isIn to the fact type Role is in FactType.

```
 1: T=2
 2: InputMetamodel=orm-feature-spec.orm
 3: TempDir=/tmp
 4: OutputDir=testset01
 5: Timeout=5
 6: PopulationSize=6 but 8 Role
 7:
 8: [Categories]
 9:    [testing.spec.ObjectTypeCardinality]
10:       Name=EntityType
11:       Range=1,2,[3-5]
12:    [End Category]
13:
14:    [testing.spec.FactTypeCardinality]
15:       Name=EntityTypePlaysRole
16:       Range=0,1,2,[3-5]
17:    [End Category]
...
```

Figure 3.3: A sample test specification for Fig. 3.2

plan using jenny. In Figure 3.3, *t* is set to 2, instructing ATIG to generate a pair-wise combinatorial test set. The `InputMetamodel` parameter (line 2) specifies the file system path to the feature specification. The `TempDir` and `OutputDir` parameters (lines 3-4) provide paths to the directories that should be used to store intermediate files and the final generated test set, respectively. The `Timeout` parameter (line 5) sets the maximum number of seconds to spend attempting to find an instance of the feature specification for a given test frame. The final parameter, `PopulationSize` (line 6), sets an upper bound on the population size of any object type in the feature specification. The parameter bounds the search for an instance that satisfies a test frame, ensuring that it terminates. In the example, each object type in the model has a maximum population size of 6 except the Role object type, which may have up to 8 members.

Choosing an appropriate population size is important to ensure that an effective test set is generated in a reasonable amount of time. On one hand, when the population size is set too high, the problem of identifying a population that fits a test frame becomes intractable, particularly if no such population exists. The search must exhaustively cover the space of inputs within the population size to determine that no satisfying population exists. On the other hand, when the population size is set too low, the search for a population that fits a test frame will often fail when such a population does exist, because the population is larger than the maximum population size. The tester may find an appropriate population size and timeout settings for a given feature and test specification by trial and error.

The remainder of a test specification defines the categories and choices that are used to partition the input space in order to generate test inputs. Figure 3.3 shows two categories that might be defined for the feature specification in Figure 3.2. ATIG presently supports just one kind of category, called a *cardinality category*. A cardinality category partitions the input space according to the population size of a specific object type or fact type from the feature specification. Figure 3.3 shows a cardinality category definition for an entity type (lines 9-12) and a fact type (lines 14-17). Each cardinality category refers to the name of an object or fact type from the feature specification, and includes a partition of the possible values (non-negative integers) into choices. In practice, the choices need not form a proper partition of the non-negative integers, or even of the possible values up to the global scope. Line 11 defines three choices for the "EntityType" category, grouping ORM models into those that have 1 entity type, those that have 2 entity types, and those that have between 3 and 5 entity types inclusive.

Many kinds of categories other than cardinality categories could no doubt be defined, and other category kinds may be added to ATIG in the future. Presently, ATIG restricts the tester to cardinality categories for three reasons. First, experience indicates that cardinality categories are sufficient to ensure that diverse test sets are generated. Second, they are simple to implement. Third, the restriction to cardinality categories makes possible an optimization that in many cases significantly reduces the amount of time necessary to generate a test set. The optimization is discussed in Section 3.2.2.

3.1.3 ATIG Outputs

ATIG returns a set of populations of the feature specification, one population for each test frame in the generated test plan. It stores a population as sample population data within a copy of the original feature specification, enabling the tester to view the population using NORMA's sample population editor. To transform a generated population into a test input in the format expected by the model transformation to be tested, the tester must create a conversion script. The effort required to produce this conversion script depends primarily on the details of the input format expected by a model transformation. If the format is a simple plain-text representation, the conversion is straightforward. A conversion script for more complex representations, in particular for graphical source notations that include diagram information, might require a significant time investment. For example, to convert a sample population of the feature specification from Figure 3.2 into a file that can be loaded in NORMA as an ORM model, a conversion script must translate each sample population into the corresponding XML representation used by NORMA, and should additionally include diagram layout information. The tester should determine, for a given model transformation, whether or not the effort necessary to create a conversion script is warranted. If only a handful of small test sets will be generated for a given feature specification, it may be more practical to perform the conversion from sample population to test input by hand.

3.2 ATIG Internals

ATIG uses an iterative algorithm to identify *likely-infeasible choice combinations* automatically. We call a choice combination likely-infeasible when we have verified that no corresponding valid population exists *up to a certain size.*

Figure 3.4: The ATIG Test Generation Algorithm

```
$allloySpec$ := orm2alloy($featureSpec$)
$infeasChoiceCombos := \emptyset$
**repeat**
    $testFrames$ := jenny($testSpec, infeasChoiceCombos$)
    **for** $i := 1$ to count($testFrames$) **do**
        $alloySpecWithFrame := alloySpec +$ choices2alloy($testFrames[i]$)
        $alloyPopls[i]$ := alloy($alloySpecWithFrame, tspec.popsize$)
        **if** $\bot = alloyPopls[i]$ **then**
            **for all** $choiceCombo \in 2^{testFrames[i]}$ sorted ascending by size **do**
                $alloySpecWithChoices := alloySpec +$ choices2alloy($choiceCombo$)
                **if** $\bot =$ alloy($alloySpecWithChoices, tspec.popsize$) **then**
                    add($infeasChoiceCombos, choiceCombos$)
                    **break**
            **break**
**until** $testFrames$ contains no infeasible choice combinations
**for** $i := 1$ to count($testFrames$) **do**
    $ormPopls[i]$ := alloy2orm_popl($alloyPopls[i]$)}
```

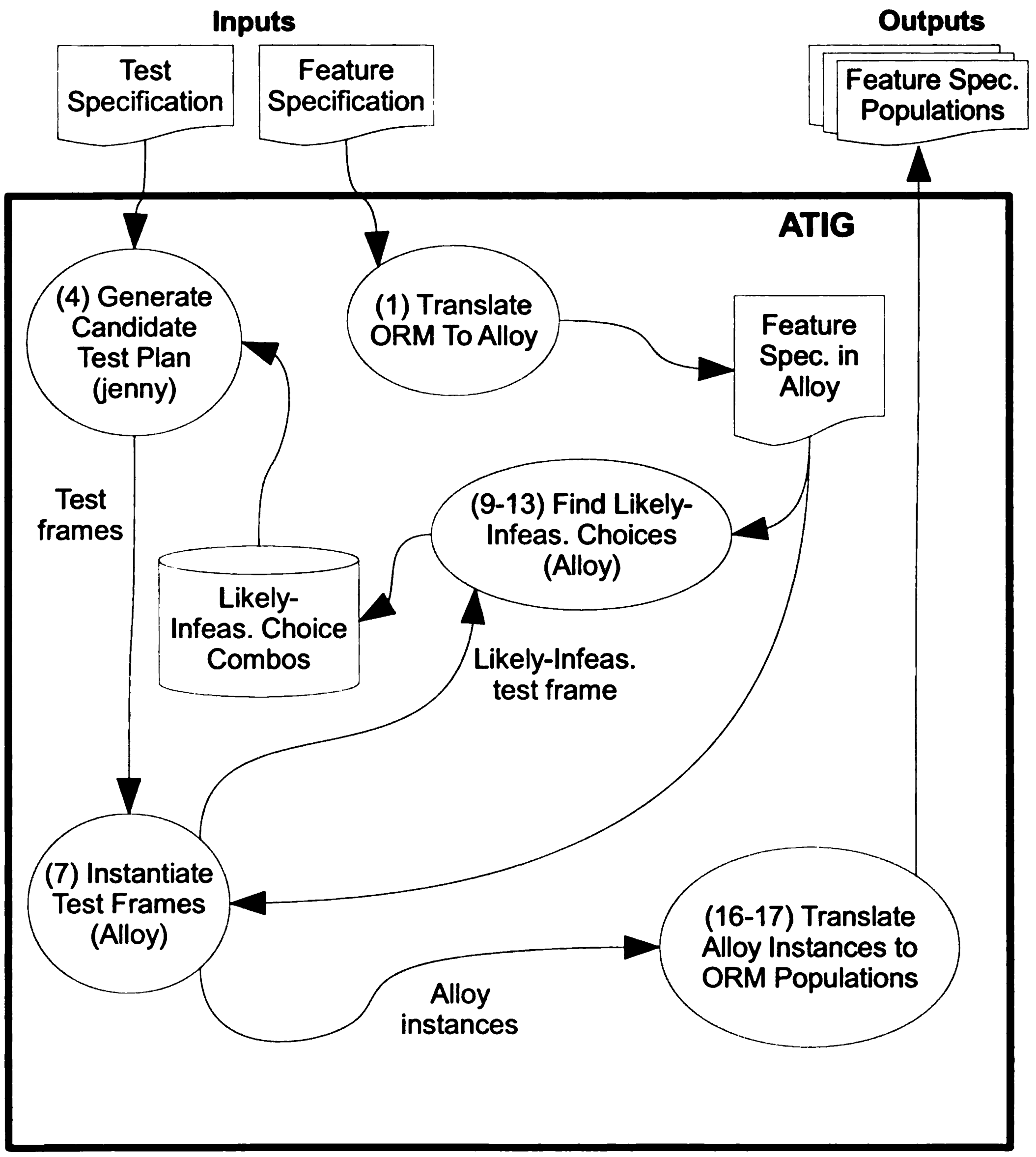

Inputs
Test Specification
Feature Specification
Outputs
Feature Spec. Populations
ATIG
(4) Generate Candidate Test Plan (jenny)
(1) Translate ORM To Alloy
Feature Spec. in Alloy
Test frames
(9-13) Find Likely-Infeas. Choices (Alloy)
Likely-Infeas. Choice Combos
Likely-Infeas. test frame
(7) Instantiate Test Frames (Alloy)
(16-17) Translate Alloy Instances to ORM Populations
Alloy instances

Figure 3.5: The data flow between modules in ATIG

3.2.1 Algorithm Description

Figure 3.4 shows the ATIG algorithm, and Figure 3.5 shows a complimentary view of how data flows between the steps in the algorithm.

ATIG first translates the feature specification into Alloy (Figure 3.4, line 1). The body of the algorithm (lines 3-15) loops until a test plan with no infeasible test frames has been created.

Within this loop, ATIG first calls jenny (line 4) to construct a t-way combinatorial test plan. ATIG passes jenny the following items from the test specification[5]:

- the value of t,
- a sequence containing the number of choices in each category, and
- an encoding of all likely-infeasible choice combinations that have been discovered.

jenny returns the test plan as a set of encoded test frames. Each test frame is encoded as a number-letter pair, where the number indicates a category and the letter indicates a choice. Categories are numbered increasing from 1. Choices are encoded as letters, increasing alphabetically from "a". For example, consider the example test specification from Figure 3.3. For this specification, the letter-number pair "2c" refers to a choice of 2 for the population size of the EntityType plays Role fact type from the feature specification in Figure 3.2.

ATIG next attempts to generate a population of the feature specification that fits each encoded test frame (lines 5-14). First, ATIG translates an encoded test

[5]For simplicity, in Figure 3.4, I show the test specification as an argument to jenny.

frame into a set of Alloy constraints (line 6), one constraint for each choice. The example choice "2c" from above would be translated into an Alloy constraint on a relation that represents the EntityType plays Role fact type. ATIG then appends the constraints to the Alloy feature specification (line 6), and invokes Alloy Analyzer on the result (line 7). Alloy Analyzer searches within the population size selected by the user for a population of the feature specification that fits each choice in the frame. If Alloy Analyzer finds a satisfying population, ATIG moves on to the next frame.

If not[6], ATIG loops through choice subsets in the test frame in increasing size order (lines 9-13) to find the smallest choice subset that has no satisfying population within the maximum population size. Because the subset has no satisfying population, no test frame containing the subset will have a satisfying population either. Thus, ATIG stores the subset as a likely-infeasible choice combination (line 12), and returns to the top of the main loop, where a new test plan is generated (line 4).

When a test plan containing no infeasible choice combinations has been generated, ATIG converts the Alloy populations into populations of the original feature specification (lines 16-17). Each converted population is saved in a separate copy of the feature specification, and can be viewed using NORMA.

3.2.2 Cardinality Analysis

The algorithm in Figure 3.4 relies solely on computationally expensive calls to Alloy Analyzer to detect likely-infeasible choice combinations via exhaustive search. To

[6] I denote a return value of "unsatisfiable" in Figure 3.4 as $\perp$

reduce the number of Alloy Analyzer invocations, ATIG uses an extension of Figure 3.4. At the start of the algorithm, ATIG analyzes the feature specification to identify relations that must hold among the population sizes of its object and fact types. The relations form a system of inequalities, each of the form $x \leq y_1 y_2 \ldots y_i$ or $x \leq y_1 + \ldots + y_i$. Before each Alloy Analyzer invocation (lines 7 and 11), ATIG uses the current choice combination and the maximum population size to initialize conservative upper and lower bounds on the variables in the inequalities. ATIG substitutes the bounds for the variables in the right-hand side of each inequality to calculate improved upper bounds. The substitution is repeated until either a fixpoint is reached, or an upper bound crosses a lower bound. In the former case, nothing is learned and ATIG invokes Alloy Analyzer as in Figure 3.4. In the latter case, there is no satisfying solution to the inequalities that agrees with the choice combination. Thus, the choice combination is invalid, and there is no need to invoke Alloy Analyzer.

The algorithm above is a simple adaptation of an algorithm by Smaragdakis *et al.* [24] for determining whether a model expressed in a special subset of ORM (termed ORM$^-$) can be populated. Their algorithm is based on a proof that a system of inequalities constructed in a particular way (detailed in the paper) from an ORM$^-$ model has a solution if and only if the model can be populated. They prove that the algorithm is guaranteed to terminate in polynomial time. In my adaptation of the algorithm, a solution to the system of inequalities does *not* imply that a valid population fitting the current choice combination exists, because the feature specification may use ORM features that are not in the ORM$^-$ subset.

However, the converse still holds. Any valid population of an ORM model M is also a valid population of the model M' obtained by removing all non-ORM$^-$ features. Thus, if there are no valid populations of M', there are also no valid populations of the original model M.

The algorithm only applies to choices that bound the population sizes of object and fact types in the feature specification. At present, ATIG supports only cardinality categories; therefore, the algorithm is automatically applicable to all choice combinations.

3.2.3 Practicality of the Algorithm

Two issues call into question the practicality of the ATIG algorithm.

1. Smaragdakis et al. [24] have shown that finding a valid population of an ORM model is NP-Hard. The search space quickly explodes as the size of ORM models increase. Yet the ATIG algorithm depends on repeated searches for ORM populations that fit individual test frames.

2. In the worst case, ATIG invokes Alloy Analyzer 2^n times on all subsets of a test frame in order to identify a minimal likely-infeasible choice combination, where n is the number of categories in the test specification. Therefore, identifying a likely-infeasible choice combination could quickly become intractable as the number of categories increases.

These issues place practical limits on the sizes of the feature and test specifications, and on the size of the generated test inputs. A tester must construct the feature specification carefully to include the modeling features thought most likely to expose errors while remaining small enough to analyze with Alloy Analyzer. If a feature specification is too large to analyze, the tester may have to abstract away important details of the source notation. Too much abstraction may reduce the likelihood that a generated test set will expose errors. Similarly, the number of categories must remain small enough that all subsets of a test frame may be checked in a reasonable amount of time, if necessary.

The next chapter describes a case study in which ATIG is used to generate a test set for an industrial code generator of moderate size. The results of the case study suggest that the ATIG algorithm is a practical solution for generating model transformation test inputs.

Chapter 4

Case Study

I conducted a case study to investigate two questions:

1. Can ATIG generate test inputs for a real-world model transformation in a reasonable amount of time?

2. Does the adequacy of test inputs generated with ATIG compare favorably with that of manually created test inputs?

In the study, I used ATIG to generate a test set for VisualBlox, a model transformation developed at LogicBlox, Inc. I used the generated test set to look for errors in all available VisualBlox releases. While the study was not sufficiently broad to answer either of the above questions definitively, the results lend anecdotal support in favor of the hypothesis that ATIG can generate adequate test sets for realistic model transformations. The modest success of the study warrants further development of ATIG, and a more thorough investigation of its usefulness. The study also led to insights regarding how ATIG might be improved.

VisualBlox automatically transforms conceptual data models (expressed in ORM) into database schema definitions to streamline the development of database applications. VisualBlox is implemented as a set of XML Schema Language Transformations (XSLT) templates. Briefly, each template describes how to generate database code from some ORM modeling feature[1]. During development, programmers manually created an ad hoc test set that exercises all ORM features supported by VisualBlox. However, no effort was made to systematically test features in combinations.

In the case study, I used ATIG to generate a test set of ORM models that systematically cover combinations of ORM language features. First, I created a feature specification that describes the ORM modeling features to be tested. Next, I created a test specification that partitions the space of input models according to categories and choices defined in terms of the feature specification. I then used ATIG to generate a set of populations of the feature specification, each representing a valid ORM model. To obtain the final set of test inputs, I wrote a conversion script to translate each generated population into an ORM file in the XML format that VisualBlox consumes. Finally, I executed VisualBlox for each test input to obtain database code, and executed that code in a fresh database. Unexpected error or warning messages, from VisualBlox or from the database engine, were considered test failures. I also manually inspected the generated database code for errors.

The following sections describe the case study in more detail. Section 4.1 describes the feature specification used to generate ORM test models for VisualBlox.

[1]In reality, there is not a one-to-one relation between modeling language features and templates. A modeling feature may have several associated templates, and a template may be invoked for more than one modeling feature. Additionally, some templates are used to generate (or generate code from) intermediate representations.

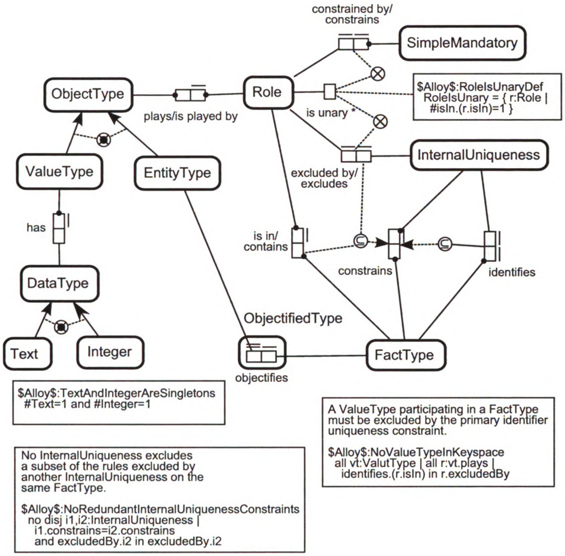
Section 4.2 describes how the test specification was created. In Section 4.3, I describe the populations that ATIG produced from the feature and test specifications. In Section 4.4, I describe how I translated each population into a valid VisualBlox input. Section 4.5 describes the results of running the generated test set on several versions of VisualBlox. Finally, Section 4.6 discusses some lessons learned from the case study and its results.

4.1 Feature Specification

VisualBlox translates ORM models into schema declarations in the Datalog^{LB} language. Because VisualBlox translates ORM models, ATIG must generate a suite of ORM models selected to cover some set of features of the ORM language itself. Thus, for this case study, the feature specification describes the structure and static semantics of ORM models.

To simplify exposition, I divided The ORM feature specification from the case study into five diagrams, Figures 4.1 through 4.5. Figure 4.1 shows the core ORM language features, including roles, fact types, entity types, value types, simple mandatory constraints and internal uniqueness constraints.

Derivation rules and complex static semantic constraints are expressed textually, in Alloy, and embedded in the feature specification as model notes. For example, in ORM it is illegal for one internal uniqueness constraint to subsume another on the same fact type. This rule is expressed as a model note with an accompanying Alloy



```
$Alloy$:RoleIsUnaryDef
 RoleIsUnary = { r:Role |
  #isIn.(r.isIn)=1 }
```

```
$Alloy$:TextAndIntegerAreSingletons
 #Text=1 and #Integer=1
```

A ValueType participating in a FactType must be excluded by the primary identifier uniqueness constraint.

```
$Alloy$:NoValueTypeInKeyspace
 all vt:ValutType | all r:vt.plays |
  identifies.(r.isIn) in r.excludedBy
```

No InternalUniqueness excludes a subset of the rules excluded by another InternalUniqueness on the same FactType.

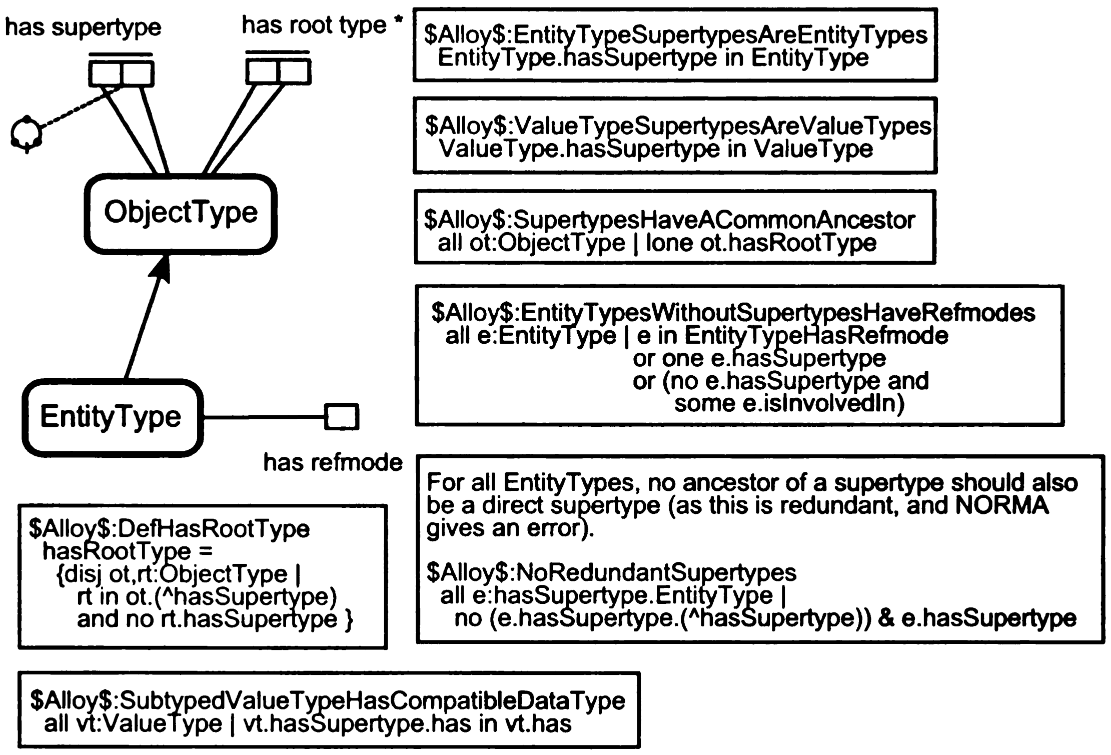
```
$Alloy$:NoRedundantInternalUniquenessConstraints
 no disj i1,i2:InternalUniqueness |
  i1.constrains=i2.constrains
  and excludedBy.i2 in excludedBy.i2
```

Figure 4.1: Core ORM modeling features

constraint, shown in the bottom-left corner of Figure 4.1[2].

Figure 4.2 and Figure 4.3 describe subtyping and objectification respectively. These features are particularly important to test in combinations, because they interact subtly with the other ORM modeling features.

[2]Alloy constraints in the feature specification must use the names that ATIG generates for the Alloy sets and relations representing object and fact types. Sets representing object types use the object type name directly. Relations representing fact types use the reading text, excluding role player names, with spaces removed and all but the leading word capitalized. Thus, in Figure 4.1, `InternalUniqueness` refers to the object type of the same name, and `excludedBy` refers to Role excluded by InternalUniqueness.



```
$Alloy$:EntityTypeSupertypesAreEntityTypes
  EntityType.hasSupertype in EntityType
```

```
$Alloy$:ValueTypeSupertypesAreValueTypes
  ValueType.hasSupertype in ValueType
```

```
$Alloy$:SupertypesHaveACommonAncestor
  all ot:ObjectType | lone ot.hasRootType
```

```
$Alloy$:EntityTypesWithoutSupertypesHaveRefmodes
  all e:EntityType | e in EntityTypeHasRefmode
                     or one e.hasSupertype
                     or (no e.hasSupertype and
                         some e.isInvolvedIn)
```

```
$Alloy$:DefHasRootType
  hasRootType =
   {disj ot,rt:ObjectType |
     rt in ot.(^hasSupertype)
     and no rt.hasSupertype }
```

For all EntityTypes, no ancestor of a supertype should also be a direct supertype (as this is redundant, and NORMA gives an error).

```
$Alloy$:NoRedundantSupertypes
  all e:hasSupertype.EntityType |
   no (e.hasSupertype.(^hasSupertype)) & e.hasSupertype
```

```
$Alloy$:SubtypedValueTypeHasCompatibleDataType
  all vt:ValueType | vt.hasSupertype.has in vt.has
```

Figure 4.2: Static semantics of ORM subtyping

Figure 4.4 describes the static semantics associated with an *intersection predicate*, which is a short-hand notation supported by VisualBlox. Simply, an intersection predicate stands for a set of role players shared by many fact types. An intersection predicate is drawn as an objectified fact type with a special annotation; we represent it in the feature specification as a unary fact type on ObjectifiedType. An intersection predicate must satisfy the usual static semantic constraints for objectification, as well as the constraints in Figure 4.4. Because intersection predicates are not a part of the ORM language, we guess that they are likely to interact with other ORM features in unexpected ways.

Finally, Figure 4.5 describes how ORM elements may be organized into groups within a model. Grouping is a feature of the NORMA modeling tool, rather than of

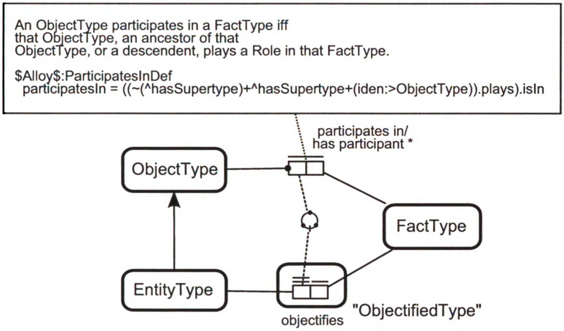
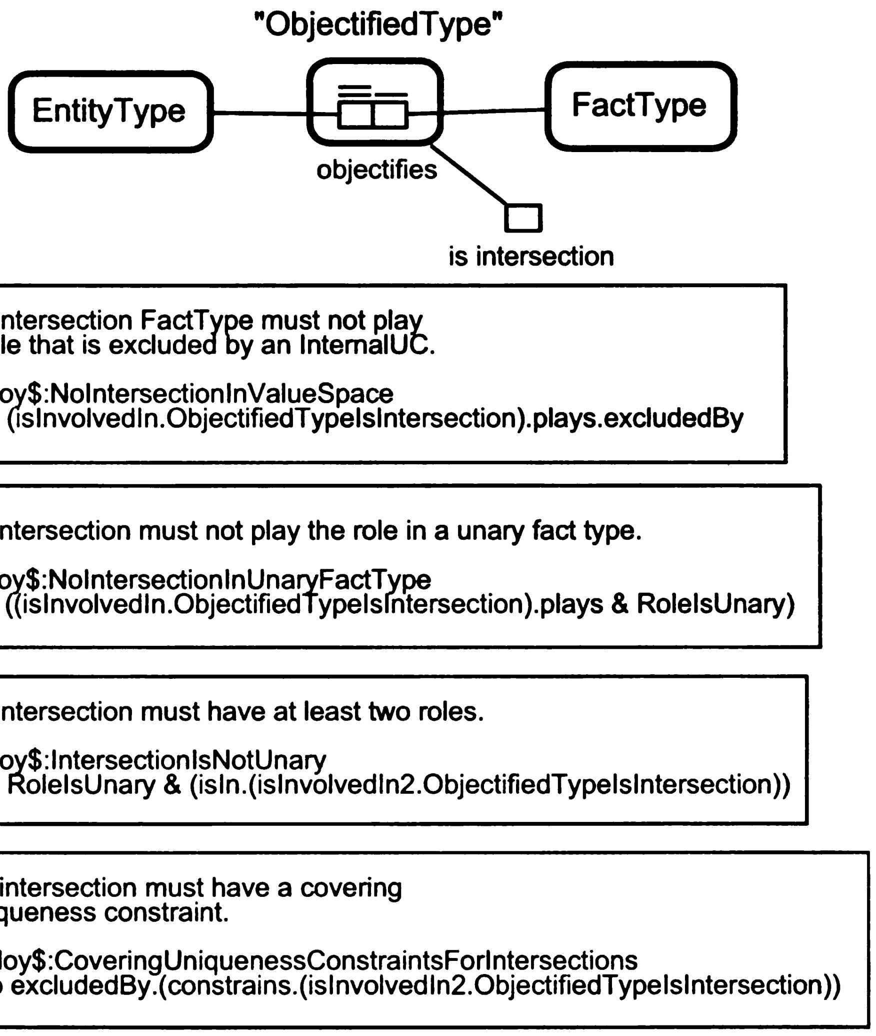

An ObjectType participates in a FactType iff
that ObjectType, an ancestor of that
ObjectType, or a descendent, plays a Role in that FactType.
$Alloy$:ParticipatesInDef
participatesIn = ((~(^hasSupertype)+^hasSupertype+(iden:>ObjectType)).plays).isIn
participates in/
has participant *
ObjectType
FactType
EntityType
objectifies
"ObjectifiedType"

Figure 4.3: Static semantics of ORM objectification

the ORM language itself. We include grouping in the feature specification because VisualBlox uses group information to organize code into namespaces.

This feature specification of ORM does not cover all ORM features. We have two reasons for modeling just a subset of ORM. First, VisualBlox does not support all of ORM. By excluding unsupported modeling features, we ensure that VisualBlox should fully translate all generated test inputs into database code. Second, a feature specification that includes all ORM features, or even the full subset of features supported by VisualBlox, would be too large to analyze with the Alloy Analyzer. Generally speaking, feature specifications must be carefully tuned to include the features thought most likely to interact in subtle ways, without causing the state space to explode to an unmanageable size. The feature specification in the case study includes the most commonly used core ORM features, as well as those features considered most likely to expose subtle interaction errors in VisualBlox.

Of course, a feature specification that precludes ill-formed and unsupported input models precludes testing that VisualBlox provides correct error messages. To test error handling with ATIG, we could include a description of unsupported modeling features in the feature specification.



An intersection FactType must not play a role that is excluded by an InternalUC.

```
$Alloy$:NoIntersectionInValueSpace
  no (isInvolvedIn.ObjectifiedTypeIsIntersection).plays.excludedBy
```

An intersection must not play the role in a unary fact type.

```
$Alloy$:NoIntersectionInUnaryFactType
  no ((isInvolvedIn.ObjectifiedTypeIsIntersection).plays & RoleIsUnary)
```

An intersection must have at least two roles.

```
$Alloy$:IntersectionIsNotUnary
  no RoleIsUnary & (isIn.(isInvolvedIn2.ObjectifiedTypeIsIntersection))
```

An intersection must have a covering uniqueness constraint.

```
$Alloy$:CoveringUniquenessConstraintsForIntersections
  no excludedBy.(constrains.(isInvolvedIn2.ObjectifiedTypeIsIntersection))
```

Figure 4.4: Static semantics of intersection predicates

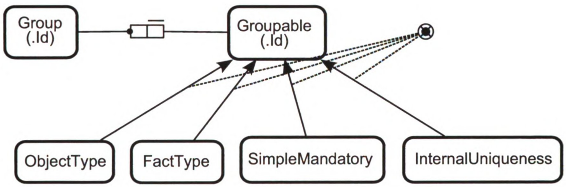


Figure 4.5: Static semantics of grouping

4.2 Test Specification

After creating the feature specification, I defined a test specification to partition the space of models for testing. I specified a t value of 2 in order to cover all pairwise choice combinations. I specified a global maximum population size of 6, with an exception for the Groupable entity type, for which I specified a maximum population size of 9. Groupable requires a larger population size because it is a supertype of several other entity types, as shown in Figure 4.5. Its population is the union of the populations of these other entity types, and so its size must be configured accordingly. I arrived at these maximum population sizes using trial and error to balance model size with analysis time. By running ATIG with various values, I found that larger sizes took undesirably long to analyze, while smaller values produced less interesting models.

The body of the test specification consists of the categories and choices that partition the input space. I used a simple strategy for defining categories: I defined a cardinality category for each asserted[3] fact type in the feature specification. This strategy is interesting because it is simple enough to be automated. ATIG could easily be extended with an option to generate a default test specification in this manner. If this simple strategy can produce adequate test sets, then it might be useful for generating a test set early in a development cycle with minimal effort. I also hope to avoid over-estimating ATIG's effectiveness based on the VisualBlox case study. An ineffective tool may provide deceptively successful results if it is configured

[3]The population of a derived fact type is entirely determined by the population of other fact types, so attempting to vary the population size of a derived fact type independently will produce many invalid choice combinations.

Figure 4.6: Test specification for the VisualBlox case study

```
# Test all combinations for every pair of categories
N=2

# ORM metamodel of ORM models
InputMetamodel=visualblox-orm.orm

# Directory to use for temporary files
TempDir=temp

# Directory in which to store generated instances
OutputDir=visualblox-testset

# Maximum population size of any object or fact type
GlobalScope=6 but 9 Groupable
JavaInstanceTranslator=testing.visualblox.AlloyInstance2ORM

ExtendedClasspath=TestInG-VisualBlox.jar

[Categories]
  [testing.spec.FactTypeCardinality]
    Name=ValueTypeHasDataType
    Ranges=0 1 2 [3,5]
  [End Category]

  [testing.spec.FactTypeCardinality]
    Name=ObjectTypePlaysRole
    Ranges=1 2 [3,5]
  [End Category]

  [testing.spec.FactTypeCardinality]
    Name=RoleConstrainedBySimpleMandatory
    Ranges=0 1 2 [3,5]
  [End Category]

  [testing.spec.FactTypeCardinality]
    Name=ObjectifiedType
    Ranges=0 1 2
  [End Category]
```

Figure 4.6: Test Specification continued

```
[testing.spec.FactTypeCardinality]
  Name=RoleExcludedByInternalUniqueness
  Ranges=0 1 2 [3,5]
[End Category]

[testing.spec.FactTypeCardinality]
  Name=InternalUniquenessConstrainsFactType
  Ranges=1 2 [3,5]
[End Category]

[testing.spec.FactTypeCardinality]
  Name=ObjectifiedTypeIsIntersection
  Ranges=0 1 2
[End Category]

[testing.spec.FactTypeCardinality]
  Name=GroupContainsGroupable
  Ranges=0 1 2 [3,5]
[End Category]

[testing.spec.FactTypeCardinality]
  Name=ObjectTypeHasSupertypeObjectType
  Ranges=0 1 2 [3,5]
[End Category]

[testing.spec.FactTypeCardinality]
  Name=EntityTypeHasRefmode
  Ranges=0 1 2 [3,5]
[End Category]
```

very carefully for a given problem. By using ATIG with a test specification that I produced according to a generic rule, I hope to gain a more realistic view of ATIG's effectiveness.

Figure 4.6 shows the resulting test specification. Each cardinality category constrains the number of times that some relationship between modeling elements occurs in an ORM model. For example, a choice of 2 for a cardinality category on the fact type **ValueType has DataType** requires ATIG to generate an ORM model with exactly two ValueTypes in the model. In this example, each value type has exactly one data type; thus, the category is equivalent to a cardinality category on **ValueType**. The specified ranges for the category ensure that ATIG generates some ORM models with no value types, some with exactly one value type, some with two value types, and some with between three and five value types. However, the category on **EntityType objectifies FactType**[4], when combined with other categories, ensures that some models include **EntityTypes** that objectify **FactTypes**, and some that do not. I partitioned each category into choices that seemed intuitively reasonable. Specifically, I selected ranges of 0, 1, 2, and [3-5] for choices in all but a few cases. I excluded a choice of 0 in some cases to avoid generating completely trivial models. For instances, the feature specification stipulates that each fact type must have an associated internal uniqueness constraint, so a choice of 0 for the category on **InternalUniqueness constrains FactType** would correspond to models with no fact types.

[4]also named **ObjectifiedType**, and referred to as such in the test specification

4.3 Generated Feature Specification Populations

ATIG generated a set of 31 populations from the feature and test specifications described in Sections 4.1 and 4.2. Each population represents a valid ORM model. Generation took 2 hours and 45 minutes on a 3.0ghz Pentium Core 2 Duo with 2 gigabytes of RAM[5]. In the process, ATIG detected 162 likely-infeasible choice combinations.

4.4 Converting Populations into ORM Models

Table 4.1: Excerpt from a population generated by ATIG
Contents of non-empty fact type in a population generated by ATIG

ObjectType plays Role	
Groupable2	Role1

Role is in FactType	
Role1	Groupable1

Role is unary
Role1

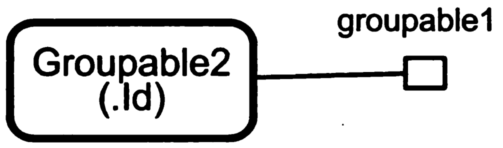


Figure 4.7: Feature specification for the case study
The ORM model corresponding to the feature spec. population in Table 4.1

I wrote a script to convert each population into an actual ORM model in the XML format processed by VisualBlox. Table 4.1 shows one of the generated populations. The conversion script extracts the population data from each file generated by ATIG, and transform it into an ORM model stored in NORMA's XML format. The script has two stages. In the first stage, the script converts population data into an XML ORM model with no diagram information. This model is sufficient to test VisualBlox

[5]ATIG does not yet take advantage of multiple cores.

(which does not interpret diagram information), but the generated test set is difficult to validate without diagram information. In the second stage, therefore, the script builds a graph representation of the ORM model, and uses Jiggle[29][6] to generate layout information. It then uses this layout information to generate a NORMA diagram, which is saved along with the model for convenient viewing. Figure 4.7 shows the ORM diagram generated from the population in Table 4.1.

4.5 Test results

I used the converted models to test all three release versions of VisualBlox.

VisualBlox Version[7]	New Errors	Old Errors	Total Errors
3.0.0	4	-	4
3.2.0	0	4	4
3.3.12	2	2	4

Table 4.2: Summary of test results for all versions of VisualBlox

The generated test set exposed previously unknown errors in all three versions of VisualBlox. Table 4.2 shows a summary of the results. "New Errors" indicates the number of errors exposed by the generated test set that were not present in earlier VisualBlox versions. "Old Errors" indicates the number of errors exposed in one version that were also exposed in earlier versions. "Total Errors" shows the total number of errors exposed by the test set.

Figure 4.8 shows a generated test input that exposed an error in the way that VisualBlox generates code for objectified fact types. This error is only exposed by models that contain objectified unary fact types. Inspection of the VisualBlox out-

[6]Jiggle is a freely available force-based graph layout library written in Java.

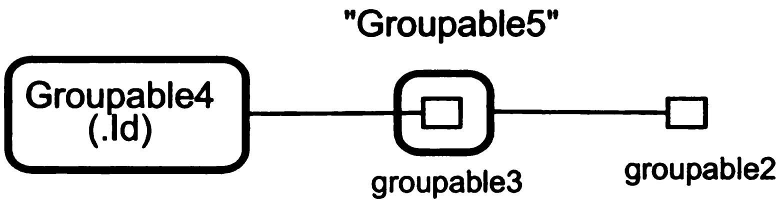


Figure 4.8: Generated test input that exposed a VisualBlox error

put for this test model revealed that VisualBlox failed to generate an important consistency constraint for objectified unary fact types. All three versions of VisualBlox exhibited this error. The existing test set for VisualBlox included models with unary fact types, and models with objectified fact types, but did not include a model that combined these two features in a way that exposes the error. The error is particularly serious because it does not cause either VisualBlox or the database engine to emit any warnings or error messages. Instead, an important database constraint is silently omitted, which could allow insertion of inconsistent data in a production database.

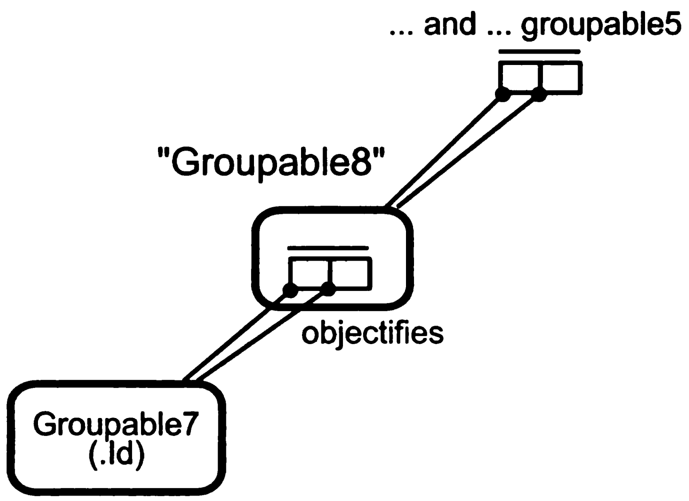


Figure 4.9: Another error-exposing test input

Figure 4.9 shows another error-exposing test input generated by ATIG. VisualBlox generates a database predicate for each role in an objectified fact type. When the same object type plays multiple roles of the same objectified fact type, VisualBlox mistakenly generates two database objects with the same name. This error was present in all three tested versions of VisualBlox. The test set manually created by VisualBlox developers did not expose it. Thus, without automated testing, this error may have persisted for a long period of time before being discovered.

4.6 Discussion

The case study demonstrated that ATIG can automatically generate valid test inputs for a real-world model transformation in a practical amount of time. VisualBlox supports a large subset of the ORM language, providing a realistic test of ATIG's practicality. Moreover, the generated inputs exposed multiple previously-unknown errors. These results, though anecdotal, warrant further development and research on ATIG.

Figure 4.10: An infeasible choice combination detected by ATIG

(1) |ObjectType plays Role| = 2
(2) |Role constrained by SimpleMandatory| = 2
(3) |EntityType objectifies FactType| = 1

Some important lessons were learned in the course of the case study.

- Manually identifying infeasible choice combinations that arise from static semantic constraints is not practical. ATIG was developed to overcome a limitation of existing combinatorial testing tools, namely that users must manually identify infeasible choice combinations. The case study illustrates why this limitation is significant when testing model transformations.

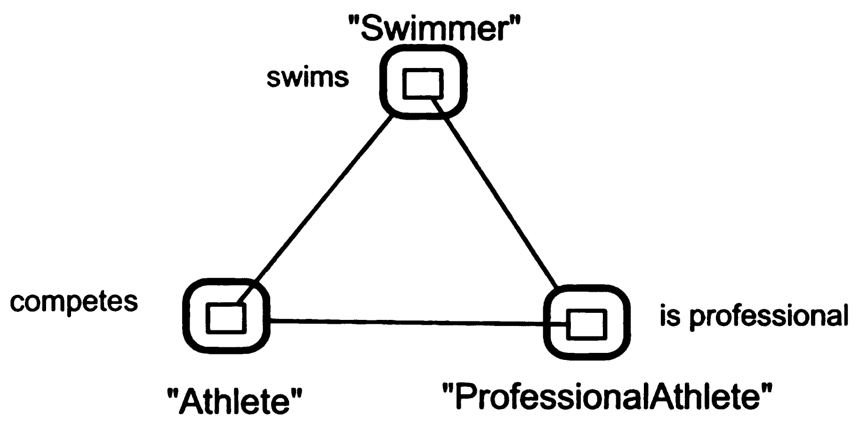


Figure 4.11: An invalid ORM model accepted by NORMA

Consider the reasoning required to determine whether the combination in Figure 4.10 is feasible or not. To fit the choice combination, a model must have exactly two roles, one objectified type, and at least one other object type to play a role in the fact type being objectified. The feature specification requires that *all* object types play at least one role, and an objectified fact type is an object type. Thus, one role must be played by the objectified type, and the remaining role must belong to it. However, now choice 2 cannot be satisfied because no unary role may be mandatory (the feature specification captures this rule with an exclusion constraint on Role is unary and Role constrained by SimpleMandatory). As there are no other alternatives that would allow mandatory constraints without violating choice 1 or 3, the choice combination is invalid.

- **Creating a feature specification may reveal cases that developers missed.** The feature specification must include a complete description of a source notation's static semantics to ensure that no invalid models are generated. During the validation of an early version of the feature specification used in the case study, ATIG generated several populations describing unusual ORM models. The models consisted of a cycle of objectified fact types: Each fact type had a role player, yet the model contained no object types. One such model is shown in Figure 4.11, with more meaningful labels added to aid comprehension. Though these models are not valid ORM models, they can currently be created in NORMA, and NORMA fails issue an error or warning message.

- **Small models are sufficient to expose subtle errors in a model transformation.** ATIG cannot generate large models as test inputs, due to the inherent difficulty of the problem. This lack of scalability precludes using ATIG to generate populations of large ORM models, such as ORM data models for database applications. However, the results of this case study indicate that small models are sufficient to expose errors in model transformations. Thus, ATIG's lack of scalability does not prevent ATIG from generating useful test sets.

- **The cost of detecting infeasible choice combinations can be amortized over many ATIG executions.** ATIG was executed repeatedly as I added to the feature specification for the case study, in order to validate the changes. I observed that likely-infeasible choice combinations typically remained infeasible when the feature specification was extended. To avoid spending time identifying the same infeasible combinations on each execution, I extended ATIG to store likely-infeasible feature combinations in a file between runs. ATIG now reloads stored likely-infeasible combinations at the start of a run, if they exist. Each reloaded combination is checked to ensure that it has not become feasible due to a change in the feature specification or the maximum population size. Checking an infeasible choice combination requires a single invocation of Alloy Analyzer, versus at worst 2^n invocations (for n categories) to find the combination in a test frame. Combinations that become feasible are discarded.

The test generation time reported in Section 4.3 was for an ATIG execution that did not make use of previously discovered infeasible choice combinations. In practice, a new test set can be generated in much less time.

Chapter 5

Related Work

The work most closely related to ATIG is divided into two categories:

- work on generating combinatorial test sets that avoid invalid inputs, and
- work on generating test inputs for model transformations.

5.1 Combinatorial testing and constraints on program inputs

A practical combinatorial testing tool must account for constraints on program inputs. Recall that a t-way test plan is a set of test frames, where each test frame describes an equivalence class of program inputs. In an efficient t-way test plan, each test frame covers at least one combination of t choices that is not covered by any other frame. Thus, each frame must describe at least one valid program input to preserve t-way choice coverage in a test set constructed according to the test plan.

Throughout this thesis, we have referred to choice combinations that describe only invalid program inputs as *infeasible* choice combinations.

Combinatorial methods and tools vary in their support for constraints. Some combinatorial testing methods, such as IPO [19] or the method described by Hnich et al [17], leave out constraint support entirely. Others offer limited constraint support. For example, the Deterministic Density Algorithm (DDA) [6] supports only *soft constraints.* A soft constraint is a feasible choice combination that the tester does not wish to appear in the generated test set, such as a choice combinations that the tester considers unlikely to expose errors. The DDA algorithm attempts to avoid soft constraints, but does not guarantee that a generated test plan will exclude them.

	Categories			
	Access	Billing	Call Type	Status
	Loop	Caller	Local	Success
Choices	ISDN	Collect	Long_Distance	Busy
	PBX	Toll-Free	International	Blocked

Table 5.1: Test specification for a telephone billing system

The TestCover tool [23], as reported by M. Cohen et al. [5], requires the user to remove infeasible choice combinations by decomposing the test specification into sub-specifications. TestCover generates a t-way test plan for each sub-specification and returns the union of these test plans. Table 5.1 shows a simple test specification for a telephone billing system, taken from [20], that we use to illustrate this method. Suppose the program specification includes the constraint that international calls are never toll-free or collect. To remove the infeasible choice combinations, a TestCover user would split the test specification into two specifications, each containing all categories. One sub-specification would include only the "Caller" choice from the Billing category, and the other would exclude the "International" choice from the "Call-Type" category.

Several combinatorial testing tools provide generate test plans that exclude all infeasible choice combinations supplied by the user. AETG [4] accepts test specifications written in the AETGSpec language [20]. AETGSpec supports constraints on choice combinations, expressed as *if-then statements.* PICT [8] also supports *if-then* constraints. jenny [28], the tool used by ATIG, requires the user to supply infeasible choice combinations as a list of tuples. M. Cohen et al. developed the prototype tool mAETG_SAT [5] to show how a SAT solver can be integrated into existing algorithms to add constraint support. mAETG_SAT is based on the original AETG algorithm [4] and accepts constraints as a list of infeasible choice combinations, much like jenny.

All of these tools assume that the user can easily identify infeasible choice combinations that arise from constraints on program inputs. This assumption generally holds when categories relate one-to-one with program parameters or configuration options and choices relate directly to ranges of input values. In Table 5.1, for example, the program inputs relate directly to the test specification, and infeasible choice combinations follow directly from constraints on the inputs. Using PICT, we would represent the constraint that international calls are never toll-free or collect with an if-then statement: "IF Call Type = International THEN Billing != Collect and Billing != Toll-Free". Using jenny, we would represent the constraint as two choice pairs (International,Collect) and (International,Toll-Free).

Generating combinatorial test sets for model transformations is difficult using existing tools, because infeasible choice combinations are difficult to identify. Most infeasible choice combinations arise from the static semantics of the source notation. However, the static semantics cannot be directly expressed in terms of the test specification. As a result, a tester cannot realistically identify infeasible choice combinations for a test specification like the one in the case study. ATIG is, to my knowledge, the only combinatorial testing tool that automatically identifies infeasible choice combinations arising from constraints on a program's inputs.

5.2 Generating Test Inputs for Model Transformations

Several recent methods for generating model transformation test inputs differ in the strategies they use to partition the input space, and in the extent to which they use knowledge about the behavior of the model transformation under test. Most methods provide only limited support for constraints that express complex static semantics.

Brottier et al.'s [2] OMOGEN tool generates test inputs for model transformations. Like ATIG, OMOGEN accepts a simplified metamodel of the source notation, which they call the *effective metamodel*. The effective metamodel is expressed as a UML class diagram. OMOGEN also accepts a partition of the effective metamodel. The partition is based on choices that specify ranges of values for each attribute and association cardinality in the effective metamodel. These choices are similar to the choices in an ATIG test specification. OMOGEN uses the effective metamodel and its partition to automatically generate test models for a model transformation.

Compared with ATIG, OMOGEN has significant limitations. For instance, OMOGEN does not process textual constraints in the effective metamodel. As a result, it may generate models that violate the source notation's static semantics. In contrast, ATIG employs Alloy to ensure that all generated models respect the static semantics. Additionally, OMOGEN requires a set of test frames[1], which the tester must derive from the partition by some other means. In contrast, ATIG uses jenny to automatically derive a t-way test set from the test specification.

Wang et al. [31] present a tool-supported method for generating test inputs directly from a model transformation to be tested and a metamodel of the source notation. Their method assumes the model transformation is written in a model transformation language that is amenable to automated analysis. The metamodel must be expressed as a UML class diagram. In the method, a tool derives a set of coverage items from the transformation and the metamodel. A coverage item defines combinations of attribute and association values to be represented in the generated test set. The tool then generates a set of models that satisfy all coverage items. The algorithm for generating models from coverage items is only sketched.

Wang et al. implemented their method with a tool that generates models for transformations written in the Tefkat [25] transformation language. The authors report that the tool often generates very large test sets containing redundant models; they are investigating pairwise testing as a means of reducing test set size. The

[1]Brottier et al. use the term *model fragments.*

method description makes no mention of support for OCL constraints. Without support for textual constraints, many of the generated models will be invalid.

At least two methods generate test models from a formal description of a code generator's behavior as a graph rewriting system. A graph rewriting system consists of graph rewriting rules. A graph rewriting rule has a left-hand side (LHS) and a right-hand side (RHS). The left-hand side is a search pattern to be matched against parts of a model. The right-hand side describes how a model that matches the LHS pattern is rewritten by the rule. In these methods, a code generator is represented as a graph rewriting system, with input and output models represented as graphs. The challenge in applying these methods is in formally describing the code generator to be tested. ATIG offers a method of generating inputs that does not require a formal specification of the model transformation to be tested.

Stürmer and Conrad [26] present a method to generate test models for a code generator, given a representation of the code generator as a graph rewriting system. Their method relies on the Classification Tree Method [13] to partition the space of models that match the LHS of a rule. They then construct one model for each class of the partition to form a test set for the rule. In later work [27] the authors automate both the derivation of a classification tree and the generation of test models from that tree.

Baldan et al. [1] present a method that uses a description of both the code generator's behavior and the structure of its inputs as graph rewriting systems. A code generator's behavior and its source notation are each described by generative graph grammars, referred to as the *optimizing grammar* and the *generating grammar* respectively. Given a set of rules from the optimizing grammar, their method attempts to generate a test case that triggers those rules. The test case is generated according to the generating grammar, ensuring that it is well formed.

Ehrig et al. [10] describe a method of deriving an instance generating graph grammar from a metamodel expressed as a UML class diagram. The grammar can then be used to generate instances of the model. The method does not currently support OCL constraints; as a result, a large proportion of models generated by the grammar are invalid. The authors sketch an extension that would support a subset of OCL. However, the subset does not include iteration over collections, which occur frequently in well-formedness rules. For example, the UML 2.0 metamodel [30] includes 5 OCL constraints on a state machine region, of which 3 contain iteration.

Chapter 6

Conclusions

This thesis explored the feasibility of extending tools for the CPM and t-way combinatorial testing to support complex constraints on test inputs, with a view towards testing model transformations. I examined prior combinatorial testing tools, and saw that they require the user to identify infeasible choice combinations manually. I developed ATIG, a prototype tool that automatically infers likely-infeasible choice combinations that arise from a formal description of test inputs. In a case study, I explored ATIG's practicality by using it to generate test inputs for an industrial code generator.

The results of the case study suggest that despite scalability issues, ATIG is a practical tool for generating model transformation test inputs. ATIG was able to generate a set of test models covering a significant subset of the features supported by VisualBlox. The test set was generated in a matter of hours, and included no invalid models. Moreover, the generated test set exposed errors in all versions of VisualBlox that were not exposed by an existing test set, which was created manually.

The case study results suggest several areas for future work. I intend to explore whether ATIG can be made more scalable. Currently, ATIG generates a complete population of a feature specification in one call to Alloy Analyzer. This method quickly becomes intractable as the feature specification grows in size. However, it may be possible to improve scalability by generating a population in several steps. I will explore ways of subdividing a large feature specification that allow sub-populations to be merged efficiently.

I also plan to use ATIG to study combinatorial testing of model transformations in more detail. For example, I will compare the cost-effectiveness of 2-way, 3-way and higher test sets versus manually created test sets. I also plan to use ATIG to test model transformations that accept other types of models, such as UML models.

Finally, I plan to improve ATIG itself. The ATIG prototype currently provides only a command-line interface. I will extend ATIG with a graphical user interface to give the tester more control over ATIG during a test generation run. I also plan to speed up ATIG test generation by parallelizing the ATIG algorithm. ATIG spends a large majority of time searching for infeasible choice combinations by considering each choice subset in a test frame. Extending ATIG to distribute this work across many processor cores or networked computers would considerably reduce test generation time.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] P. Baldan, B. Konig, and I. Stürmer. Generating test cases for code generators by unfolding graph transformation systems. *Lecture notes in computer science*, pages 194–209, 2004.

[2] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proceedings of ISSRE*, volume 6, pages 85–94, 2006.

[3] T.Y. Chen, R. Merkel, G. Eddy, and P.K. Wong. Adaptive random testing through dynamic partitioning. In *Proceedings of the Fourth International Conference on Quality Software*, pages 79–86, 2004.

[4] D.M. Cohen, S.R. Dalal, J. Parelius, G.C. Patton, and NJ Bellcore. The combinatorial design approach to automatic test generation. *IEEE software*, 13(5):83–88, 1996.

[5] M.B. Cohen, M.B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139. ACM New York, NY, USA, 2007.

[6] C.J. Colbourn, M.B. Cohen, and R.C. Turban. A deterministic density algorithm for pairwise interaction coverage. In *Proc. of the IASTED Intl. Conference on Software Engineering*, volume 41, pages 242–252. Citeseer, 2004.

[7] M. Curland and T. Halpin. Model driven development with norma. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 286a–286a, 2007.

[8] J. Czerwonka. Pairwise testing in real world. In *24th Pacific Northwest Software Quality Conference*, volume 82, 2006.

[9] R.A. DeMilli and A.J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[10] K. Ehrig, J.M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 8(4):479–500, 2009.

[11] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[12] G. Gheorghiu. A look at Selenium. *Better Software*, 7(8):38, 2005.

[13] M. Grochtmann and K. Grimm. Classification trees for partition testing. *SOFTWARE TEST VERIF RELIAB*, 3(2):63–82, 1993.

[14] T. Halpin. ORM 2. *Lecture notes in computer science*, 3762:676, 2005.

[15] T. Halpin and T. Morgan. *Information modeling and relational databases: from conceptual analysis to logical design.* Morgan Kaufmann, 2008.

[16] K.V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[17] B. Hnich, S.D. Prestwich, E. Selensky, and B.M. Smith. Constraint models for the covering test problem. *Constraints*, 11(2):199–219, 2006.

[18] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[19] Y. Lei and K.C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proceedings of the third IEEE High Assurance Systems Engineering Symposium*, pages 254–261, 1998.

[20] C. Lott, A. Jain, and S. Dalal. Modeling requirements for combinatorial software testing. In *Proceedings of the 1st international workshop on Advances in model-based testing*, page 7. ACM, 2005.

[21] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6), 1988.

[22] E. Seidewitz. What models mean. *IEEE software*, pages 26–32, 2003.

[23] G. Sherwood. http://testcover.com/pub/constex.php.

[24] Y. Smaragdakis, C. Csallner, and R. Subramanian. Scalable automatic test data generation from modeling diagrams. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 4–13. ACM New York, NY, USA, 2007.

[25] J. Steel and M. Lawley. Model-based test driven development of the tefkat model-transformation engine. In *ISSRE'04 (Int. Symposium on Software Reliability Engineering)*, pages 151–160, 2004.

[26] I. Stürmer and M. Conrad. Test suite design for code generation tools. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, pages 286–290, 2003.

[27] I. Stürmer, M. Conrad, H. Doerr, and P. Pepper. Systematic testing of model-based code generators. *IEEE Transactions on Software Engineering*, 33(9):622–634, 2007.

[28] `http://burtleburtle.net/bob/math/jenny.html`. jenny, June 2006.

[29] D. Tunkelang. JIGGLE: Java interactive graph layout environment. *Proceedings of Graph Drawing98*, 1998.

[30] v2.0 UML Superstructure Specification. http://http://www.omg.org/cgi-bin/doc?formal/05-07-04, 2004.

[31] J. Wang, S.K. Kim, and D. Carrington. Automatic Generation of Test models for Model Transformations. In *Proceedings of the 19th Australian Conference on Software Engineering, Washington*, 2008.