Testing Model Transformations: A case for Test Generation from Input Domain Models

Benoit Baudry

Testing Model Transformations: A case for Test Generation from Input Domain Models

Benoit Baudry

INRIA / IRISA
Campus de Beaulieu
35042 Rennes Cedex
bbaudry@irisa.fr

ABSTRACT: Model transformations can automate critical tasks in model-driven development. Thorough validation techniques are required to ensure their correctness. In this lecture we focus on testing model transformations. In particular, we present an approach for systematic selection of input test data. This approach is based on a key characteristic of model transformations: their input domain is formally captured in a metamodel. A major challenge for test generation is that metamodels usually model an infinite set of possible input models for the transformation.

We start with a general motivation of the need for specific test selection techniques in the presence of very large and possibly infinite input domains. We also present two existing black-box strategies to systematically select test data: category-partition and combinatorial interaction testing. Then, we detail specific criteria based on metamodel coverage to select data for model transformation testing. We introduce object and model fragments to capture specific structural constraints that should be satisfied by input test data. These fragments are the basis for the definition of coverage criteria and for automatic generation of test data. They also serve to drive the automatic generation of models for testing.

KEYWORDS: test input generation, input domain modelling, metamodel-based test generation, test adequacy criteria, combinatorial testing

1 Introduction

Model transformation is a key mechanism when building distributed real-time systems (DRES) with model-driven development (MDD). It is used to automatically perform a large number of tasks in the development of DRES. The DOC group at Vanderbilt University has extensively investigated MDD for DRES. In this context, Madl et al. (Madl *et al.* '06) use model transformations in order to apply model checking techniques on early design models, Gokhale et al (Gokhale *et al.* '08)

develop model transformations that automate deployment tasks of component-based DRES and Shankaran et al. (Shankaran *et al.* '09) use model transformations to dynamically adapt a DRES when its environment changes. The ACCORD/UML (Gérard *et al.* '00) methodology developed by CEA also makes an extensive use of model transformations for a model-driven development of DRES. Model transformations encapsulate specific steps in the development methodology and generate optimized code. Airbus develops large model transformations that automatically generate optimized embedded code for the A380 from SCADE models.

Due to the critical role that model transformations play in the development of DRES, thorough validation techniques are required to ensure their correctness. A fault in a transformation can introduce a fault in the transformed model, which, if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates further, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models. Several studies have investigated static verification techniques for model transformations. For example, Küster (Küster '06), focuses on the formal proof of the termination and confluence of graph transformation, or Anastasakis et al (Anastasakis *et al.* '07) analyze properties on a formal specification of the transformation in Alloy.

In this lecture we are interested in adapting software testing techniques to validate model transformations. In particular, we focus on the generation and qualification of test data for model transformations. To test a model transformation, a tester will usually provide a set of *test models*, run the transformation with these models and check the correctness of the result. While it is fairly easy to provide some input models, qualifying the relevance of these models for testing is an important challenge in the context of model transformations. As for any testing task, it is important to have precise adequacy criteria that can qualify a set of test data (Baudry *et al.* '09).

Model transformations specify how elements from the source metamodel are transformed in elements of the target metamodel. The source metamodel completely specifies the input domain of the transformation: the set of licit input models. In this context, the idea is to evaluate the adequacy of test models with respect to their coverage of the source metamodel. For instance, test models should instantiate each class and each relation of the source metamodel at least once. In the following we present test adequacy criteria based on the coverage of the source metamodel. We also discuss the automatic generation of test models that satisfy these criteria.

Before presenting the specific generation of test data for model transformation, we recall general techniques and current challenges for test generation from a model of the input domain. We briefly introduce category-partition testing (Ostrand *et al.* '88) and combinatorial interaction testing (Cohen *et al.* '97) as two black-box techniques for the systematic selection of a subset of values in large domains. These techniques are a specific case of *model-based testing*.

Utting et al (Utting *et al.* '07b) identify four different approaches to model-based testing: generation of test data from a domain model, generation of test cases from

an environmental model, generation of test cases with oracle from a behaviour model, generation of test scripts form abstract tests. Utting et al.'s book focuses mainly on the third approach, while in this lecture we will introduce techniques related to the first approach. Ammann et al (Ammann *et al.* '08) propose another classification of structures from which it is possible to design test cases: graphs, logic, input domain, syntax. According to this taxonomy, the techniques introduced in this lecture are related to the last two structures: design of test data from an input domain model and from a model of the syntax (e.g., the source metamodel for a model transformation).

2 Challenges for testing systems with large input domains

One important aspect of the growing complexity of software systems is that these systems tend to be more and more open to their environment. In particular, this means that many systems can operate on a very large number of information provided by the user and/or offer mechanisms for dynamic reconfiguration. In both cases, these systems are characterized by a very large domain on which they have to run. It is usually not possible to test these systems with all possible input and in all possible configurations. The challenge for test data generation is to propose criteria to systematically select a subset of data that will still ensure a certain level of trust in the system under test.

In this section we present several examples where such issues occur for test generation.

2.1 Large Set of Input data

The first category of systems that has a large input domain is the set of all programs that process a large set of data. These data can be provided by other software components or by users. Examples of these systems are all the web applications that process user input provided through a form.

Figure 1 displays an example of such a form that a user must fill in order to register to a conference online. On this simple, very usual form, there are 18 variables. Some of these variables can take an infinitely large number of values (all the fields that require a String value such as address, name, etc.), and some others have a finite domain: the combobox for states defines 72 values, 228 values for country, 4 values for special needs and a binary value for IEEE contact. In addition to the large domains for each variable, the global input domain for this page is the total number of combinations of values for each variable. This number is $72 * 229 * 4 * 2 * 14^{\#\text{String values}}$. It is important to notice that there exist some constraints between the fields that reduce the number of combinations. For example, if the country is neither Canada nor the USA, there is no need to provide a value for the Province /State field. In order to test this registration system, it is necessary to select a subset of all possible input values. In particular, it is necessary; first to reduce the

set of all possible String values to a finite set of test data; second to select a small number of configurations of data.

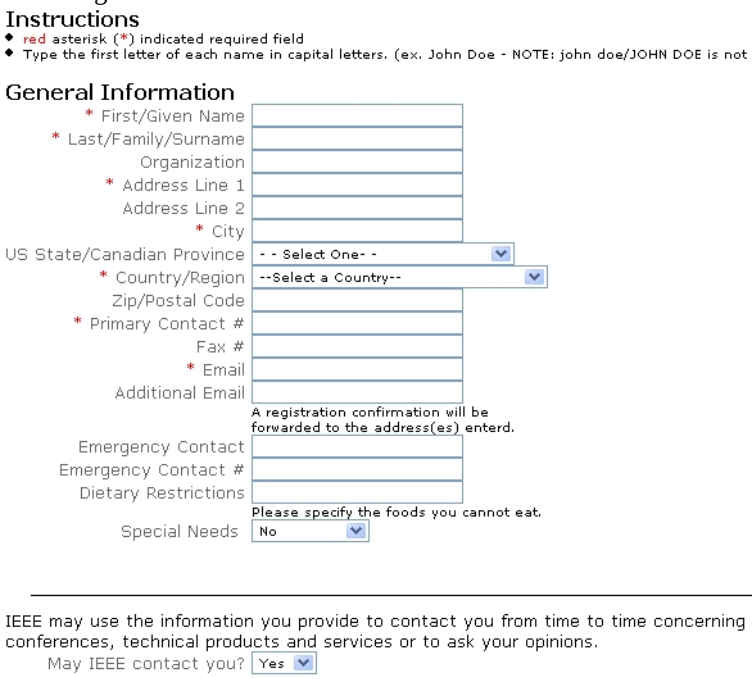

Figure 1. *An example of a large domain: a web form*

2.2 Configurable systems

Highly configurable and adaptive systems represent a second category of systems that are characterized by large domains. Microsoft Internet Explorer is an example of such a system. It has 31 configurable options on the security tab. There are around 19 trillion possible configurations for this tab (Cohen *et al.* '06) and the system should behave correctly in all these conditions. An emerging trend in embedded systems is the ability to adapt to changes in the environment at runtime. In this case, the set of all possible environment settings represents the set of all configurations under which the system is expected to work. In the context of the DiVA project (DiVA '08), the CAS company develops a customer relationship management system. The requirements for this system describe 23 environmental properties which represents 10^7 possible combinations and as much different environments to which the system is expected to adapt. The variables for configuring the system usually have a finite domain. The challenge for testing is thus to select a minimal set of configurations to test the system.

2.3 Grammarware and Model Transformations

An interesting category of system with a large domain consists of all the systems which input data is modelled with a grammar or a metamodel. The programs which input domain can be described with a grammar are known as grammarware (Hennessy *et al.* '05; Klint *et al.* '05), and usually have an infinite domain. These applications include parsers, refactoring tools, programs analyzers, etc. For example, Figure 2 displays an excerpt of the grammar for the Alloy analyzer (Jackson '06). The first rule states that a specification in Alloy is composed of a number of open and paragraph constructs. Because of the '*' symbol, there can be between 0 and an infinite number of open and paragraph. This means that the input domain for the Alloy analyzer is infinite since there can be an infinite number of specifications that conform to the rules in the grammar.

```
specification ::= [module] open* paragraph*
module ::= "module" name [ "[" ["exactly"] name ("," ["exactly"] num)* "]" ]
open ::= ["private"] "open" name [ "[" ref,+ "]" ] [ "as" name ]
paragraph ::= factDecl | assertDecl | funDecl | cmdDecl | enumDecl | sigDecl
```

Figure 2. *Excerpt from Alloy grammar*

Concerning systems which input domain is modelled with a metamodel, we call theses systems *model transformations*. They are similar to grammarware programs since their input domain is potentially an infinite set of models that are licit input data for the transformation. What is interesting with grammarware and model transformations is that their input domain is explicitly captured in a finite model that can be leveraged for the definition of test adequacy criteria and to systematically identify a finite set of test data.

Since this lecture focuses on testing model transformations, we provide a detailed example of a metamodel in the following. Figure 3 displays a metamodel for a simple class diagram modelling language. This metamodel specifies a class model as being a set of CLASSIFIERs and ASSOCIATIONs. A CLASSIFIER is either a CLASS that can have a parent CLASS, a set of ATTRIBUTEs and that can be persistent, or a PRIMITIVEDATATYPE. ATTRIBUTEs can be primary, they have a name and a type. ASSOCIATIONs have a name and destination and source CLASS.

All the concepts for this simplified class diagram language are represented by classes in the metamodel. These classes have *properties* that are either attributes of primitive type (*e.g.*, the name attribute in the ASSOCIATION class) or references to other classes. The references have a role name and a multiplicity. For example, the reference from CLASSMODEL to ASSOCIATION has the role name `association` and a multiplicity * which means that a CLASSMODEL contains a set of zero or many ASSOCIATIONs. Constraints to restrict the set of licit class models are captured by references and multiplicities on the references.

Classes and properties are usually not expressive enough to specify all constraints on the structure of the modeling language. The Object Constraint

Language (OCL) can be used to add constraints and allow us to build a more precise model of the domain. For example, Figure 4 displays additional invariants on the simple class diagram metamodel of Figure 3, expressed in OCL. The first one specifies that there must be no cycle in the parent relationship between a CLASS and another one, which means that a class cannot inherit of itself or one of its parents.

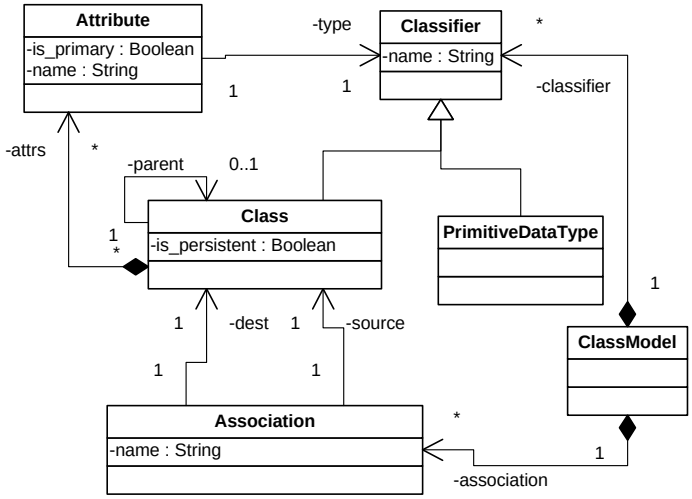

Figure 3. *Simple UML Class Diagram Metamodel*

Invariants on the metamodel

```
context Class
  inv noCyclicInheritance:
    not self.allParents()->includes(self)
  inv uniqueAttributesName:
    slef.attrs->forAll(att1,att2 | att1.name=att2.name implies att1=att2)

context ClassModel
  inv uniqueClassifierNames:
    self.classifier->forAll(c1,c2 | c1.name=c2.name implies c1=c2)
  inv uniqueClassAssociationSourceName:
    self.association -> forAll(ass1,ass2 | ass1.name=ass2.name implies
              (ass1=ass2 or ass1.src!=ass2.src))
```

Figure 4. *Additional constraints on the metamodel*

A metamodel defines the input domain of a model transformation. Thus it defines the set of models that can be passed as input to the transformations. Since the metamodel is defined as a set of classes and properties, a *model is a graph of objects*. The objects in this graph are instances of the classes defined in the

metamodel. The structure of the graph is constrained by the multiplicities on references and by all additional constraints defined on the metamodel.

The metamodel of Figure 3 models the input domain for any transformation that manipulates simple class diagrams. This metamodel can serve as a basis for the generation of a set of test data. However, the '*' on the cardinality for the set of attributes in a CLASS or the set of associations in a CLASSMODEL, means that there is potentially an infinite number of models that conform to this metamodel. More precisely, the size of the set of classes in a model is between 0 and maxInt. This cardinality alone indicates that the total number of class models that satisfy the structure defined by the metamodel can be very large. The set of classifiers in the model and the set of attributes in a class have the same multiplicity. Thus the total number of models that combine these three properties only is $maxInt^3$ which is 21 474 836 48^3 for a machine that encodes integers on 32 bits.

2.4 Testing Challenges

In all the above examples, it appears that, even with small domain models (5 classes in a metamodel or 18 fields in a web form) the number of input data and combinations of data can be very large or even infinite. It is thus impossible to test these systems with all possible data. The issue for test data generation is then to *select a subset of test data in the input domain* according to systematic criteria and that cover all relevant sub-domains in that domain.

3 Selecting test data in large domains

In this section we introduce category-partition testing (Ostrand et al. '88) and combinatorial testing strategies (Cohen *et al.* '96) that can be used separately or conjointly to systematically select a subset of test data in large input domains.

3.1 Category partition

The basic idea of category-partition testing strategies (Ostrand et al. '88) is to divide the input domain into sub-domains called *ranges*. This division is based on specific knowledge of the domain and consists in identifying subsets of values that are equivalent with respect to the behavior of the program under test. The ranges for an input domain define a partition of the input domain and thus should not overlap. Once the partitions and ranges are defined, the test generation consists in selecting one test data in each range. *Boundary testing* consists in selecting test data at the boundary of the ranges.

Definition – Partition*. A partition for a variable's domain of elements is a collection of n ranges R_1, …, R_n such that R_1, …, R_n do not overlap and the union of all subsets forms the domain. These subsets are called ranges.*

In order to use category-partition for test generation, the first step consists in identifying all the variables that define the input domain of the system under test.

These variables can be input parameters for methods, variables that represent the state of the program, environment variables, fields in a form, options to configure the system, or properties in a metamodel. Once these variables are well identified, the domain of each variable must be divided in a set of ranges that form a partition. The process of partitions construction is critical: the more the values for range boundaries are representative, the more relevant are the partitions and thus the more relevant is the test data. On the other hand, there exist no techniques to automatically identify relevant ranges.

In their introduction to software testing, Amman and Offutt (Ammann et al. '08) provide two different approaches for the identification of variables that characterize the input domain and the relevant values for partitioning this domain. The first approach is based on the interface of the system and considers all the variables in isolation. The main benefit of this approach is that it is simple and thus very straightforward to apply, but this might lead to an incomplete domain model (missing links between variables). The second approach is called the 'functionality-based approach'. In this case, the variables are identified according to the expected functionalities of the system. In particular, the variables can be identified in the requirements documents. In this case, the input domain model can be richer and thus more precise.

The construction of partitions for the domains of all variables is the key creative part of this testing approach. The boundaries for ranges in the partition capture representative values for which the system is supposed to behave in a specific way. These values are manually identified. We distinguish between knowledge-based and default partitioning. Knowledge for knowledge-based partitioning can be found

- in the requirements. For example, the requirements of the form in Figure 1 should specify the expected format of phone number or fax numbers. These formats would allow partitioning the domain of integers for these two variables.
- in the interface design. For example, in Figure 1, the variables which domain is captured in a combobox, each value in the combobox is a representative value. Here the ranges in the partition are simple ranges that contain one value each.
- in the pre and post conditions of methods if the system is deigned by contract. The values in the precondition restrict the input domain for an operation. For example, a pre condition that specifies that an integer parameter should be greater than 5, this indicates that the domain of this variable can be divided in two ranges: greater than 5 and lower or equal than 5. Similarly post conditions can provide information on ranges of values that are expected to produce results that satisfy a property.
- in the code itself. Controls on the values of input variables for the system usually capture representative values for these variables. For example, an if statement can capture a different behaviour for the system according to the value of the variable

Default partitioning can be used when no information is available on representative values. This consists in defining default ranges to partition the domain

for primitive data types. For example, the domain of strings can be partitioned in two ranges: the range that contains the empty string and the range that contains all the non-empty strings. This partition can then guide the selection of String values for the name and organization fields in the form of Figure 1. An empty string for these fields is expected to raise an exception since these fields are mandatory in this form.

In model-driven development, partition testing has been adapted to test executable UML models by Andrews et al. (Andrews *et al.* '03). They consider a class diagram, OCL pre and post conditions (OMG '03) for methods and activity diagrams to specify the behaviour of each method. From this model, the authors generate an executable form of the model, which can be tested. Dinh-Trong et al. (Dinh-Trong *et al.* '05) then propose to model test cases using UML2.0 sequence diagrams. From these test cases specifications and the class diagram, they generate a graph that corresponds to all possible execution paths defined in the different scenarios. The authors then use test criteria defined in (Andrews et al. '03) and automatically generate test data and an initial system configuration to cover each execution path.

In section 4 we show how we adapted category-partition for the definition of test coverage criteria on metamodels (Fleurey *et al.* '07).

3.2 Combinatorial interaction testing

We have seen how category partition is a possible technique to reduce infinite domains of variables in a finite set of ranges in which the variables should take at least one value. When all the variables have a finite domain (either using category-partition or because the domain is an enumeration), there remains one issue for the selection of test input: the selection of a subset of all possible combinations of variables. As we have seen in the example of the web form or of adaptable systems, an important factor for the explosion of the size of the input domain is the number of combination of variables. In this section we introduce *combinatorial interaction testing* (CIT) (Cohen et al. '97; Cohen et al. '96) as a possible approach to select a subset of all combinations while still guaranteeing a certain level of coverage.

CIT is based on the observation that most of the faults are triggered by interactions between a small number of variables (Kuhn *et al.* '04). This has led to the definition of pairwise testing, or 2-way testing. This technique samples the set of all combinations in such a way that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to t-way testing which samples the input domain to cover all t-way combinations.

For example, let us consider a simple model for a cashier at a movie theatre. The variables and the possible values are summarized in Table 1. There are 4 types of clients, three periods with different fares, three types of guidance for movies (G for no restriction, PG13 for guidance for children below 13 and R for restriction for children below 17) and three payment methods.

There are 128 possible combinations of values with all the four variables in this simple example. Pairwise testing suggests selecting a subset of all combinations in which all the combinations of pairs of variables are present. A possible solution for

pairwise testing with our movie example is displayed in Table 1. All the pairs of variables are present, but there are only 12 combinations of input data. This solution is generated by the TConfig tool provided by Alan Williams (Williams '08).

Table 1. *Input domain for movie cashier*

	Client	**Period**	**Parental Guidance**	**Payment**
Possible values	Child	Week	G	Cash
	Adult	Week-end	PG-13	Debit card
	Senior	Holiday	R	Credit card
	Student			

Table 2. *Pairwise data for movie cashier*

Client	**Period**	**Parental Guidance**	**Payment**
Child	week	G	Cash
Child	week-end	PG13	Debit card
Child	Holiday	R	CreditCard
Adult	Week	PG13	Credi card
Adult	Week-end	R	Cash
Adult	Holiday	G	Debit card
Senior	Week	R	Debit card
Senior	Week-end	G	Credit card
Senior	Holiday	PG13	Cash
Student	Week	G	Cash
Student	Week-end	PG13	Debit card
Student	Holiday	R	Credit card

The generation of T-way CIT is based on the generation of a mathematical object called a *covering array*.

Definition – Covering array*. A covering array CA (N; t, k, v) is a N × k array on v symbols with the property that every N × t sub-array contains all ordered subsets of size t from v symbols at least once.*

From the definition of a covering array, the strength t of the array is the parameter that allows achieving 2-way (pairwise), 3-way or t-way combinations. The k columns on this array correspond to all the variables in the input domain. As it is defined here, all the variables in the array must have the same number v of possible values. Since variables usually do not have the same number of values (e.g., the variables in the movie cashier example have 3 or 4 values), there exist a more general structure called a *mixed level covering array*. This array also has k columns, but the variables of each column do not necessarily have the same number of values.

The problem of generating a minimal covering array for a set of variables is a complex optimization problem that has been studied in a large number of work (Cohen et al. '97; Cohen *et al.* '08; Shiba *et al.* '04; Williams *et al.* '01). There exist

several tools that implement these solution for CIT automatic generation (Czerwonka '08; Utting *et al.* '07a).

It is important to notice that there exist very few work that have tackled the automatic generation of CIT in the presence of constraints between variables. In our example, there are at least two combinations in Table 1 that should raise an exception in the system: the second combination tests the system with a child who sees a PG13 film and the third combination tests it with a child who sees an R film. In order to include properties that forbid combinations of these values, CIT generation techniques have to allow the introduction of constraints in the algorithms that generate covering array. Recent work by Cohen et al tackles this specific issue (Cohen et al. '08).

4 Metamodel-based test input generation

Models in model-driven development are productive assets for the development of software systems. This means that models are built in such a way that they can be automatically manipulated by programs. The structure and semantics of models are captured in a metamodel and the programs that manipulate models are referred to as model transformations. The metamodel is thus a model of the input domain for a model transformation.

In Fleurey et al (Fleurey et al. '07), we have proposed several coverage criteria over a metamodel in order to select and qualify a set of models for testing. These criteria are based on the notion of object and model fragments that define constraints on objects and models that must be present in a set of models adequate for testing. The models that serve as test data for a model transformation are called *test models*.

In this section we introduce how we have adapted category-partition on metamodels to limit the input domain for test models. Then we define the notions of object and model fragments used to define coverage criteria. We also discuss possible strategies to automatically generate models that satisfy these criteria.

4.1 Metamodel coverage criteria

In section 2.3 we showed that the size of the domain for a model transformation can be very large because of * multiplicities for some properties of the metamodel. In order to restrict the size of the space that has to be explored for test models generation, we define partitions on the domain and/or multiplicity of each property in a metamodel.

Notation – Default Partition. *The default partitions for primitive data types are noted as follows:*

- *Boolean partitions are noted as a set of sets of Boolean values. For example,{false}} designates a partition with two ranges: a range which contains the value true and a range which contains the value false*

- *Integer partitions are noted as a set of sets of Integer values. For example, {{0}, {1}, {x | x ≥ 2}} designates a partition with three ranges: 0, 1, greater or equal to 2.*
- *String partitions are noted as a set of sets of String values. A set of string values is specified by a regular expression. For example {{“”}, {“.+”}} designates a partition with two ranges: a range which contains the empty string and a range which contains all strings with one or more character. In the regular expression language, “.” designates any character and “+” specifies that the preceding symbol can be repeated once or more.*

Figure 5 displays default partitions and ranges for the simple class diagram metamodel of Figure 3 (partitions on the multiplicity of a property are denoted with #). These default partitions, based on the types of properties, are automatically generated for any metamodel by the MMCC tool (MMCC '08). Yet, if there are other representative values in the context of the transformation under test, the tester can enrich the partitions to ensure that they are used in the test models.

Attribute::is_primary	{true}, {false}
Attribute::name	{« »}, {.+}
Attribute::#type	{1}
Class::is_persistent	{true}, {false}
Class::#parent	{0}, {1}
Class::#attrs	{0}, {1}, {x | x>1}
Association::name	{« »}, {.+}
Association::#dest	{1}
Association::#source	{1}
ClassModel::#association	{0}, {1}, {x | x>1}
ClassModel::#classifier	{0}, {1}, {x | x>1}

Figure 5. *Partitions for the simple CD metamodel*

Based on this partitioning we can define a simple test adequacy criterion that specifies that each range in each partition should be covered by one test model at least. For example, the {{“”}, {“.+”}} for the name attribute of ASSOCIATION specifies that there should be one test model that contains an ASSOCIATION with any name that is a non-empty String and another ASSOCIATION that has an empty name. Similarly, the {{0}, {1}, {x | x > 1}} partition for the multiplicity of the association reference of CLASSMODEL specifies that there should be a test model that has no association, another model that exactly one association and another model that has more than 1 association.

Stronger adequacy criteria should require specific combinations of values or ranges of values. A naïve strategy would consist in requiring one test model for each combination of ranges, but even in the simple case of the class diagram language, this would mean generating 1296 test models. This represents a very large number of models considering the small number of concepts in the metamodel. We thus have defined criteria that limit the number of combinations that have to be covered while ensuring the coverage of the metamodel (Fleurey et al. '07).

4.2 Model and object fragments for test adequacy criteria

A model, instance of a metamodel, can be seen as graph of objects that are instances of the classes in the metamodel. The adequacy criteria on a set of test models are defined as constraints on the objects in a test model. We capture the notion of constraint on one object in an *object fragment*. An object fragment constrains the values of certain properties by specifying in which range the property should take its value. It is important to note that an object fragment does not necessarily define constraints for all the properties of a class, but can partially constrain the properties (like a template).

In order to define constraints on the combination of object fragments in complete models, we define the notion of *model fragment*. A model fragment is a collection of object fragments. A model fragment is a constraint that should be satisfied by one test model.

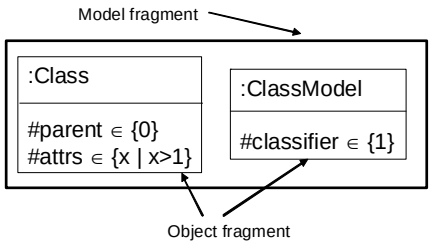

Figure 6. *Example of object and model fragment*

Figure 6 displays an example of a model fragment that includes two object fragments. One object fragment [Class::is_persistent {true} Class::#attrs {x | x>1}] specifies that there should be an instance of CLASS in one test model such that the property is_persistent takes its values in the range {true} and the property attrs has a multiplicity in the range {x | x>1}. There is no constraint on the multiplicity of the parents property of the object. The second object fragment specifies that there should be a CLASSMODEL such that the number of classifiers is in the range {1}. There is no constraint on the multiplicity of the association reference of CLASSMODEL. The model fragment specifies that there should be one test model that contains two objects that satisfy both object fragments.

The test adequacy criteria for test models are defined as a set of model fragments that combine ranges of values for the properties according to different strategies. Each criterion specifies a set of model fragments that should be satisfied by a set of test models in order to fulfil the criterion.

Test criterion for metamodel coverage *: A test criterion specifies a set of model fragments for a particular source metamodel. These model fragments are built to guarantee class and range coverage as defined in the following rules.*

Rule 1*- Class coverage: Each concrete class must be instantiated in at least one model fragment.*

Rule 2 *-Range coverage: Each range of each partition for all properties of the metamodel must be used in at least one model fragment.*

Test criterion satisfaction for a set of test models*: A set of test models satisfies a test criterion if, for each model fragment MF, there exists a test model M such that all object fragments defined in MF are covered by an object in M. An object O corresponds to an object fragment OF if, for each property constraint in OF, the value for the property in O is included in the range specified by OF.*

The weakest coverage criteria we propose are called AllRanges and AllPartitions. They both ensure range coverage by combining property constraints in two different manners. *AllRanges* enforces the two rules defined above. *AllPartitions* is a little stronger, as it requires values from all ranges of a property to be used simultaneously in a single test model.

In a metamodel, properties are encapsulated into classes. Based on this structure and on the way metamodels are designed, it is natural that properties of a single class have a stronger semantic relationship with each other than with properties of other classes. To leverage this, we propose four criteria that combine ranges class by class. These criteria differ on the one hand by the number of ranges combinations they require and on the other hand by the way combinations are grouped into model fragments. The formal definition of the four criteria CombΣ, ClassΣ, CombΠ, ClassΠ is provided in (Fleurey et al. '07).

We have built a metamodel (Fleurey et al. '07) that formally captures the notions of partition for properties in a metamodel, of object and model fragment. This metamodel is the basis for the construction of the MMCC tool (MMCC '08). MMCC can generate partitions for the properties of a metamodel, compute the set of object and model fragments according to an adequacy criterion and check whether a set of models satisfies the criterion.

4.3 Discussion

A first important point that has to be noted is that the criteria defined previously are based only on the MOF description of the metamodel. However, the input domain of a transformation is usually modeled with additional constraints. For examples, the constraints of Figure 4 restrict the set of possible class diagrams. The pre condition displayed in Figure 7 further restricts the input domain of a model transformation with a pre condition on the class diagram metamodel. Since the definition of model and object fragments does not consider these constraints, some test criteria will require a model fragment in which there is only one class and that this class has only one attribute which in not primary. However this contradicts the pre condition of the transformation. Thus, the test criteria might specify uncoverable model fragments. This is a general issue with test adequacy criteria: they define some objectives that cannot be satisfied by any test case. For example, structural test criteria for programs specify infeasible paths (Adrio *et al.* '82), or mutation analysis produces equivalent mutants (Offutt *et al.* '97).

```
pre atLeastOnePrimaryAttribute:
  input.attrs -> select(att1|att1.is_primary)->size()>=1
```

Figure 7. *Pre condition for a transformation on class diagrams*

Another point worth mentioning is the similarities between the coverage criteria on a metamodel and some criteria that have been studied to generate test data for grammarware programs. Amman et Offutt (Ammann et al. '08) propose simple criteria to ensure 'terminal symbol coverage' and 'production coverage' which are very close to the simplest criteria for metamodel coverage: instantiated each metaclass and each association between these classes. Once these minimal criteria are satisfied by data that cover a grammar, more complex criteria consist in combining complex terms to form larger data that test the interactions between rules. In that case, there is the same combinatorial issue than for metamodels. Lämmel et al (Lämmel *et al.* '06) directly address this issue and propose 'control mechanisms' to limit the explosion. Hennessy et al (Hennessy et al. '05) study different strategies to limit this explosion and compare them in terms of code coverage and fault detection. In Baudry et al (Baudry *et al.* '05) we proposed a technique driven by mutation analysis in order to limit the generation of test data to data that can kill mutants.

4.4 Automatic synthesis of test models

We can expect several benefits from automatic generation of test models. This can save time and effort during the development of a model transformation. This can help when the transformation evolves or the source metamodel changes to take new concepts into account. It can also assist the manual construction of test models with a tool that automatically completes a model to make conform to the metamodel.

There are two major challenges for the automatic generation of test models

1. Heterogeneous constraints. The constraints that define the input domain and coverage criteria are defined by different actors using different languages. The metamodel is defined by language designers, the restrictions on a metamodel for a specific transformation are defined by transformation developers, test criteria and test objectives are defined by testers. These different models and constraints are expressed with various formalisms: EMOF and OCL for the metamodel, OCL or patterns for the restriction on the input domain, model fragments for test criteria.
2. Automatic constraint solving. Adequate test models are defined by a large set of constraints that have to be considered as a whole in order to generate a correct model.

In Sen et al.(Sen *et al.* '08; Sen *et al.* '07) we proposed to transform all the different constraints to a common formalism compatible with automatic constraints solving techniques. First we proposed a methodology using *constraint logic programming*. We present a transformation from a metamodel, constraints,

fragments and a partial model to a constraint logic program (CLP). We solve/query the CLP to obtain value assignments for undefined properties in the partial model. In a second approach (Sen et al. '08) we proposed to combine all constraints in an Alloy model. Alloy is a lightweight formal modelling language that allows automatic analysis. In particular it is connected to several SAT solvers that can automatically solve a set of constraints and generate instances in the search space.

Other approaches tackle the automatic generation of models to test a model transformation. Two constructive approaches propose to generate models first and check the constraints afterwards. Brottier et al. (Brottier *et al.* '06) consider only the class diagram definition of the source metamodel to generate objects and assemble them according to adequacy criteria based on model fragments in order to build complete models. Ehrig et al. (Ehrig *et al.* '06) analyze the metamodel to generate rules that create instances of all non-abstract classes and links between the instances. The major limitation of these approaches is that they do not consider all the additional constraints on the input domain in the generation process. As a consequence, a large number of generated models do not satisfy the complete set of constraints and thus the transformation cannot process them for testing.

5 Conclusion

Automatic model transformations are essential assets in the model-driven development of embedded systems. In this lecture we have focused on testing as a possible approach to assess their quality. In particular we have presented the issues related to the selection of test models in the large space of input models for a transformation.

We have first discussed general issues related to testing systems characterized by large domains. We have introduced category-partition and combinatorial interaction testing (CIT) as two existing techniques to limit the combinatorial explosion test data in the presence of a very large input domain. Then, we have focused on the notion of model and object fragments to define test criteria on a metamodel that models the input domain of a model transformation. These criteria aim at selecting test models in the possibly infinite set of input models for a transformation. We also briefly discussed the challenges for automatic generation of models according to these criteria and possible solution.

There remain many challenges for an efficient selection of test data in large domains. Concerning CIT we mentioned that it is important to integrate constraints between variables in the generation of testing configurations in order to obtain licit and meaningful combinations. Similarly, the automatic generation of test models must integrate all the constraints on the input domain in order to generate models that can be processed by the transformation. A related issue is the definition of a precise model for the input domain: it is important that all constraints are captured when building this model to allow automatic analysis and effective test generation. Automatic generation of test models also faces usual issues for automatic test generation: interpretation of the test data, management of these data (priorities,

regression testing selection, etc.). More generally, there remain important issues for model transformation testing (Baudry et al. '09). The work presented here on selection criteria is a necessary step towards a global solution.

Acknowledgements. This work has been partially supported by the European project DiVA (EU FP7 STREP). I am also extremely grateful to Franck Fleurey, Jean-Marie Mottu and Sagar Sen whose PhD work has largely contributed to the understanding of model transformation testing as presented here and to Yves Le Traon for the numerous fruitful discussions on software testing.

6 References

(Adrio et al. '82) Adrio, W. R., M. A. Branstad and J. C. Cherniavsky. "Validation, Verification, and Testing of Computer Software", ACM Computing Surveys. 14 (2), 1982: 159 - 192.

(Ammann et al. '08) Ammann, P. and J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008

(Anastasakis et al. '07) Anastasakis, K., B. Bordbar and J. M. Küster, "Analysis of Model Transformations via Alloy". In Proceedings of MoDeVVa'07 in conjunction with MODELS'07. October 2007. Nashville, TN, USA. pp

(Andrews et al. '03) Andrews, A., R. France, S. Ghosh and G. Craig. "Test adequacy criteria for UML design models", Software Testing, Verification and Reliability. 13 (2), 2003: 95 -127.

(Baudry et al. '05) Baudry, B., F. Fleurey, J.-M. Jézéquel and Y. Le Traon. "From Genetic to Bacteriological Algorithms for Mutation-Based Testing", Software Testing, Verification and Reliability. 15 (1), 2005: 73-96.

(Baudry et al. '09) Baudry, B., S. Ghosh, F. Fleurey, R. France, Y. Le Traon and J.-M. Mottu. "Barriers to Systematic Model Transformation Testing", Communications of the ACM. Accepted for publication, 2009.

(Brottier et al. '06) Brottier, E., 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'06 (Int. Symposium on Software Reliability Engineering). 2006. Raleigh, NC, USA. pp 85 - 94

(Cohen et al. '97) Cohen, D. M., S. R. Dalal, M. L. Fredman and G. C. Patton. "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Transactions on Software Engineering. 23 (7), 1997: 437 - 444.

(Cohen et al. '96) Cohen, D. M., S. R. Dalal, J. Parelius and G. C. Patton. "The combinatorial design approach to automatic test generation", IEEE Software. 13 (5), 1996: 83 - 88.

(Cohen et al. '08) Cohen, M. B., M. B. Dwyer and J. Shi. "Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach", IEEE Transactions on Software Engineering. 34 (5), 2008: 633-650.

(Cohen et al. '06) Cohen, M. B., S. Joshua and G. Rothermel, "Testing across configurations: implications for combinatorial testing". In Proceedings of Workshop on Advances in Model-based Software Testing (A-MOST). November 2006. Raleigh, North Carolina, USA. pp 1-9

(Czerwonka '08) Czerwonka, J. (2008). "Pairwise testing tools." Retrieved December, 2008, from http://www.pairwise.org/tools.asp.

(Dinh-Trong et al. '05) Dinh-Trong, T., N. Kawane, S. Ghosh, R. France and A. Andrews, "A Tool-Supported Approach to Testing UML Design Models". In Proceedings of ICECCS'05. June 2005. Shanghai, China. pp 519 - 528

(DiVA '08) DiVA. (2008). "DiVA EU FP7 STREP." Retrieved December, 2008, from http://www.ict-diva.eu/.

(Ehrig et al. '06) Ehrig, K., J. M. Küster, G. Taentzer and J. Winkelmann, "Generating Instance Models from Meta Models". In Proceedings of FMOODS'06. June 2006. Bologna, Italy. pp 156 - 170

(Fleurey et al. '07) Fleurey, F., B. Baudry, P.-A. Muller and Y. Le Traon. "Towards Dependable Model Transformations: Qualifying Input Test Data", Software and Systems Modeling. 2007.

(Gérard et al. '00) Gérard, S., N. S. Voros , C. Koulamas and F. Terrier, "Efficient System Modeling for Complex Real-Time Industrial Networks using the ACCORD/UML Methodology". In Proceedings of DIPES'00. 2000. Paderborn, Germany. pp 11 - 22

(Gokhale et al. '08) Gokhale, A., K. Balasubramanian, J. Balasubramanian, A. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons and D. C. Schmidt. "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications", Elsevier Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture. 73 (1), 2008: 39 - 58.

(Hennessy et al. '05) Hennessy, M. and J. P. Power, "An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software". In Proceedings of ASE'05. November 2005. Long Beach, CA, USA. pp 104 - 113

(Jackson '06) Jackson, D., Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006

(Klint et al. '05) Klint, P., R. Lämmel and C. Verhoef. "Toward an engineering discipline for grammarware", ACM Transactions on Software Engineering and Methodology. 14 (3), 2005: 331 - 380.

(Kuhn et al. '04) Kuhn, D. R. and D. D. Wallace. "Software Fault Interactions and Implications for Software Testing", IEEE Transactions on Software Engineering. 30 (6), 2004: 418 - 421.

(Küster '06) Küster, J. M. "Definition and Validation of Model Transformations", Software and Systems Modeling. 5 (3), 2006: 233 - 259.

(Lämmel et al. '06) Lämmel, R. and W. Schulte, "Controllable combinatorial coverage in grammar-based testing". In Proceedings of TestCom 2006. May 2006. New York City, USA. pp 19 - 38

(Madl et al. '06) Madl, G., S. Abdelwahed and D. C. Schmidt. "Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking", Real-time Systems Journal. 33 (1), 2006: 77 - 100.

(MMCC '08) MMCC. (2008). "Metamodel coverage checker." Retrieved December, 2008, from http://www.irisa.fr/triskell/Softwares/protos/MMCC/.

(Offutt et al. '97) Offutt, A. J. and J. Pan. "Automatically Detecting Equivalent Mutants and Infeasible Paths", Software Testing, Verification and Reliability. 7 (3), 1997: 165 - 192.

(OMG '03) OMG. (2003). "UML 2.0 Object Constraint Language (OCL) Final Adopted specification." 2005, from http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.

(Ostrand et al. '88) Ostrand, T. J. and M. J. Balcer. "The category-partition method for specifying and generating functional tests", Communications of the ACM. 31 (6), 1988: 676 - 686.

(Sen et al. '08) Sen, S., B. Baudry and J.-M. Mottu, "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing". In Proceedings of ICST'08 (International Conference on Software Testing Verification and Validation). April 2008. Lillehamer, Norway. pp 328-337

(Sen et al. '07) Sen, S., B. Baudry and D. Precup, "Partial Model Completion in Model Driven Engineering using Constraint Logic Programming". In Proceedings of INAP'07 (International Conference on Applications of Declarative Programming and Knowledge Management). October 2007. Würzburg, Germany. pp

(Shankaran et al. '09) Shankaran, N., J. Kinnebrew, X. Koutsoukos, C. Lu, D. C. Schmidt and G. Biswas. "An Integrated Planning and Adaptive Resource Management Architecture for Distributed Real-time Embedded Systems", IEEE Transactions on Computers, Special Issue on Autonomic Network Computing. 2009.

(Shiba et al. '04) Shiba, T., T. Tsuchiya and T. Kikuno, "Using Artificial Life Techniques to Generate Test Cases for Combinatorial Testing". In Proceedings of COMPSAC '04: 28th Annual International Computer Software and Applications Conference. 2004. Washington, DC, USA. pp 72--77

(Utting et al. '07a) Utting, M. and B. Legeard. (2007a). "MBT Tools." Retrieved December, 2008, from http://www.cs.waikato.ac.nz/~marku/mbt/CommercialMbtTools.pdf.

(Utting et al. '07b) Utting, M. and B. Legeard, Practical Model-Based Testing, Morgan Kaufmann, 2007b

(Williams '08) Williams, A. (2008). "TConfig - Test configuration generator." Retrieved December, 2008, from http://www.site.uottawa.ca/~awilliam/TConfig.jar.

(Williams et al. '01) Williams, A. and R. L. Probert, "A Measure for Component Interaction Test Coverage". In Proceedings of ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2001). June 2001. Beirut Lebanon. pp 304-311