Acknowledgement

«The discovery of nuclear reactions need not bring about the destruction of mankind any more than the discovery of matches.»
Albert Einestein

Contents

Résumé en français
(French Summary)

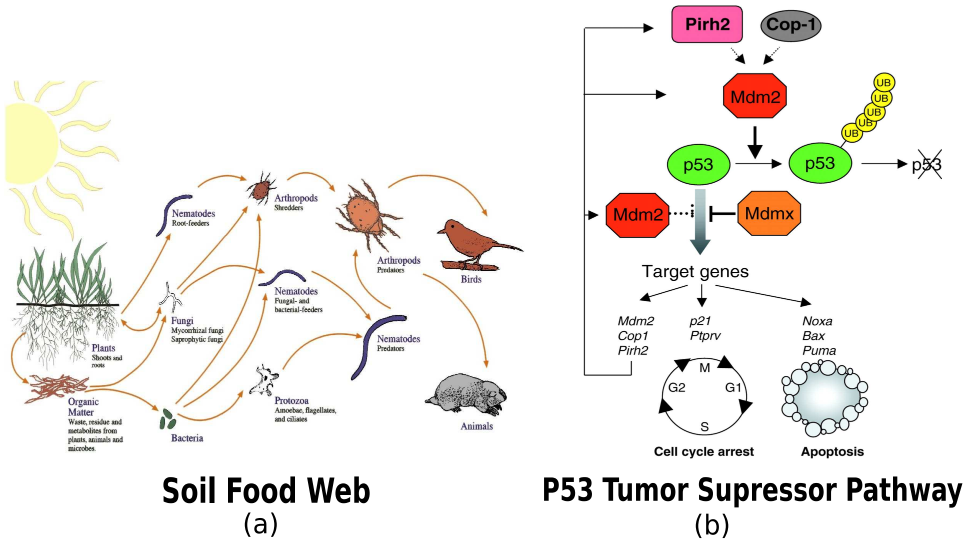
Chapter 1

Introduction

Model-driven engineering (MDE) is an approach to specify, construct, validate and maintain complex software systems using first class artifacts called *models*. MDE has emerged from a number of areas in software development such as object-oriented analysis and design languages, object-oriented methodologies [24] [73] [127], and Computer-Aided Software Engineering (CASE) endeavours in the 80s and 90s to automate several steps in software engineering [113] [27].

Models are *graphs of inter-connected objects* in a *modelling domain*. A modelling domain defines a *set of models* where each model is constructed using a common set of concepts and relationships. For instance, in this thesis we consider the specification of two modelling domains: (a) *metamodels* that specify a set of models in a modelling language (b) *feature diagrams* or *feature models* that specify a set of product models or simply products in a Software Product Line (SPL). Very often the creation of useful or *effective models* in a modelling domain require the satisfaction of constraints from heterogeneous sources. For instance, creating a workflow model for a business process using the well-known Unified Modelling Language (UML) activity diagram requires the model to satisfy UML well-formedness rules, business logic, economic constraints, quality of service constraints, and security restrictions. Human modellers with experience incrementally create such effective models by tacitly ensuring that the models are *correct by construction* and satisfy constraints from heterogenous sources. Still and all, this process is extremely tedious and sometimes impossible if there is a need to create thousands of models. **Can we automate the creation of effective models given the heterogenous sources of knowledge?** This is the question that intrigues us and the subject of this thesis.

The introduction is organized as follows. The notion of effective model discovery situates itself in the *global context* of discovering effective structures in science and engineering. We briefly describe this global context in Section 1.1. This thesis addresses the problem of automating discovery in the contemporary and specific context of MDE which we describe in Section 1.2. A number of scenarios in MDE necessitate generation of effective models. Our motivation stems from these scenarios that we describe in Section 1.3. In Section 1.4, we present the general *problem context* and its challenges. We present our thesis and describe the methodology for automatic effective model and product discovery in Section 1.5. We enlist the contributions of our thesis in Section 1.6. Finally, we present the organization of the thesis in Section 1.7.



Soil Food Web
(a)

P53 Tumor Supressor Pathway
(b)

Figure 1.1: Effective Structures in Scienctific Discovery: (a) Soil Food Web (b) Tumor Suppression Pathway

1.1 Discovery of Effective Structures in Science and Engineering

Scientific discovery often culminates into representing structure in nature as *networks of entities* or *graphs of objects*. For instance,

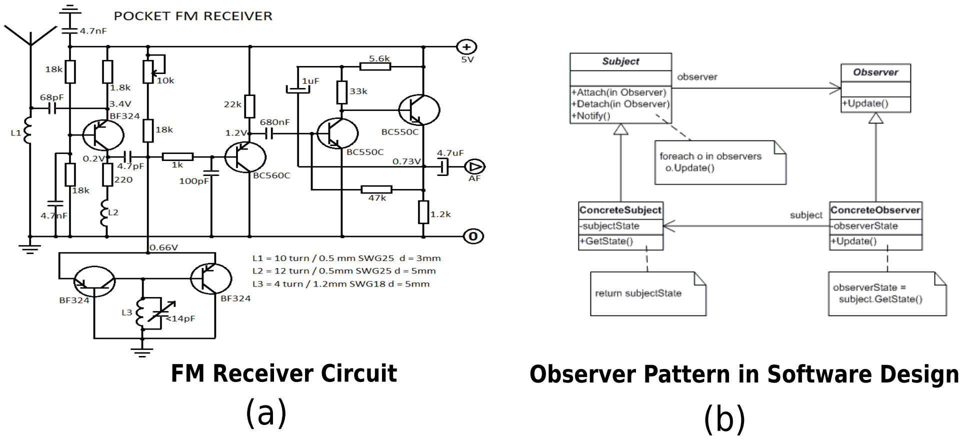
- **Food webs** are representations of the predator-prey relationships between species within an ecosystem or habitat. A common example is the *soil food web* shown in Figure 1.1. The soil food web is often found in a garden bio-compost.

- **Biochemical reaction networks** or metabolic pathways represent vital molecular exchanges in living beings. The widely studied *tumor suppressor pathway* shown in Figure 1.1 (b) illustrates the crucial role of protein p53 in cell death. Cell death is important in order to regulate cancerous growth.

Knowledge from experiments, data analysis and mental tacit lead to the discovery of such *effective structures* in nature. The existence of effective structures is not limited to the virtuosity of nature. We humans are endowed with the ability to represent and create effective structures such as buildings, bridges, robots, and complex software.

Design in engineering often results into representing effective man-made structures as graphs of objects. For instance,

- **Electronic circuits diagrams** represent a network of electrical components that achieve a given purpose. The *FM Receiver Circuit* shown in Figure 1.2, for instance, is used in millions of radio devices.



FM Receiver Circuit

(a)

Observer Pattern in Software Design

(b)

Figure 1.2: Effective Structures in Engineering: (a) FM Radio Circuit (b) Observer Design Pattern

- **Software Design Patterns** represent general reusable solutions to commonly occurring problems in software design. They are often represented as *class diagrams* in object-oriented software engineering.The *observer pattern* in Figure 1.2 (b) is a common pattern in software requiring distributed event handling. The well-known photo editing program Adobe Photoshop is one such software product.

Very much like discovery in science, design in engineering is guided by knowledge from a number of sources coupled with the creativity of an engineer. Can this process of scientific discovery or design in engineering using various sources of knowledge be *automated*? This question has been a subject of study for several decades with the advent of the modern computer.

Computer programs have been used to discover structure in nature. For instance, inspired by Karl Popper's logic of scientific discovery [123], Pat Langley, Herbert Simon, G. Bradshaw, and Zytkow developed several computer programs such as Bacon, Glauber, Dalton, and Stahl described in their book [94]. These programs were guided by heuristics to successfully rediscover historical laws in chemistry.

Evolutionary computing approaches have been develop to automate design in engineering such as generation of electronic circuits [88]. Computer programs implementing an evolutionary approach contest for the "Humies Award" conferred each year at the GECCO conference. The award of $10,000 is given to the approach with most human-competitive results. In the software engineering community, recent conferences such as the Automated Software Engineering (ASE) conference provide competitive venues for presenting approaches to generating software structures.

In this thesis, we address the question of automatic discovery in the contemporary context of Model-driven Engineering of complex software systems.

1.2 Context: Model Driven Engineering

MDE [110] aims to grease the wheels of complex software creation using first class artifacts called *models*. The MDE philosophy is using models to represent important artifacts in a system such as requirements, high-level designs, data structures, views, interfaces, model transformations, test cases, and implementation-level artifacts such as source code. A model is constructed in a *modelling domain* that captures a set of common concepts and relationships. The construction of a model in a modelling domain may be further constrained using well-formedness rules and constraints from various sources.

The general notion of a modelling domain can be *specialized* in many ways. A precise specification of concepts and relationships that defines a set of models is a modelling domain. For instance, a *metamodel* specifies the modelling domain of a modelling language. The well-known Unified Modelling Language (UML) modelling language [116] has its own metamodel that specifies the set of all UML models. Another, example of a modelling domain is a *feature diagram* or *feature model* that specifies a set of products in a Software Product Line(SPL). Models in a modelling domain can be loaded/stored, manipulated, and transformed to other models/implementation artifacts to solve software problems.

MDE provides a number of software processes and technologies to allow creation of modelling domains and the transformation of its models. Historically, the Model-driven Architecture (MDA) trademark marketed by the Object Management Group (OMG), presents a model-driven approach to system development. The MDA approach begins development of a modelling domain for platform-independent models (PIMs), which are incrementally transformed or refined into lower-level platform specific models (PSMs) in another modelling domain. The PSMs are reified into implementation artifacts such as implementation code. This automatic construction of systems from high-level models allows software engineering expertise to be captured as reusable *model transformations* applied more reliably and efficiently. Currently, the widely accepted framework for specifying modelling domains is the Eclipse Modeling Framework (EMF) [58]. For instance, metamodels are created in the EMF *Ecore* format to specify the domain of a modelling language. Model transformation [142] languages such as the imperative Kermeta [82] [108], rule-based ATL [76] [75] [3], graph grammar based AToM^3 [67], Viatra [156] transform models. Model transformation languages are expected to conform to the Query-View-Transformation (QVT) standard [75]. Different types of model transformations can be created using these languages as classified in [44]. Model transformations may transform models within the same modelling domain (endogenous transformations), between different modelling domains (exogenous transformations) and even realize the classical view of generating executable code from a high-level model.

Our focus in this thesis is the automatic discovery or computer-assited discovery of models in a modelling domain.

1.3 Motivation: Why the Need for Automatic Model Discovery?

Our motivation for automatic discovery in the general context of model-driven engineering stems from existing computational discovery endeavors in heterogenous domains. Computational dis-

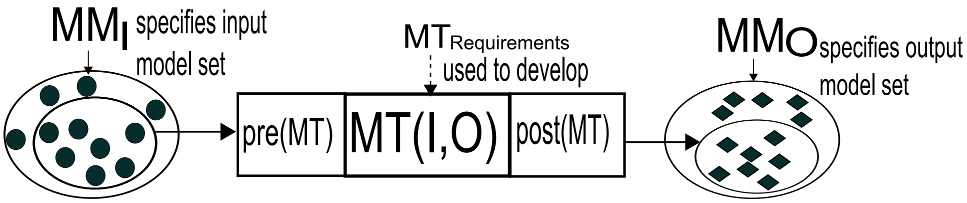

MMI specifies input model set
MTRequirements used to develop
MMO specifies output model set
pre(MT)
MT(I,O)
post(MT)

Figure 1.3: A Model Transformation

covery approaches in these domains range from systems biology [33] [126], to engineered physical systems [97] [132] [56], [88]. We see automatic discovery of effective models in a modelling domain as general framework subsuming existing approaches to effective structural discovery in heterogeneous areas. MDE of software systems is no exception. In this thesis, we investigate three scenarios in MDE as described below:

1.3.1 Scenario 1: Test Generation for Model Transformations

Model transformations are core software artifacts in MDE. A simple model transformation MT takes input models conforming to an input metamodel MM_I and produces output models conforming the output metamodel MM_O as shown in Figure 1.3. Not all models specified by the input metamodel can be processed by the model transformation. Therefore, we compose pre-conditions $pre(MT)$ that restrict some models from being processed by the model transformation. The output models must also satisfy a set of constraints called the post-condition $post(MT)$. The model transformation itself is built using knowledge from a set of requirements $MT_{Requirements}$.

Testing a model transformation requires input model that can detect bugs in the transformation MT. Manually creating such test models is tedious since they must be graphs of objects that must conform to MM_I, $pre(MT)$, and use information from $MT_{Requirements}$. Manual creation becomes impossible when we need to create thousands of such test models that encode different test objectives. Therefore, there is a clear need to automate the generation of test models that satisfy knowledge from various sources such as MM_I, $pre(MT)$, and use information from $MT_{Requirements}$. The automatic generation of input models exalts to the level of automatic discovery of test models if we validate that they can indeed detect bugs in a transformation. We can qualify the effectiveness of test models via techniques such as *mutation analysis* for model transformations [107]. Based on a description of this scenario, we ask *how do we generate test models and qualify their bug detecting effectiveness*?

1.3.2 Scenario 2: Partial Model Completion in a Model Editor

Modellers often use model editors to incrementally build models. For instance, the TopCaseD editor [54] can be used build UML models as shown in Figure 1.4. The model shown is an incomplete UML state machine. The model does not have an initial state which violates a well-formedness rule. There are infinite possible ways to complete the model such that it becomes a

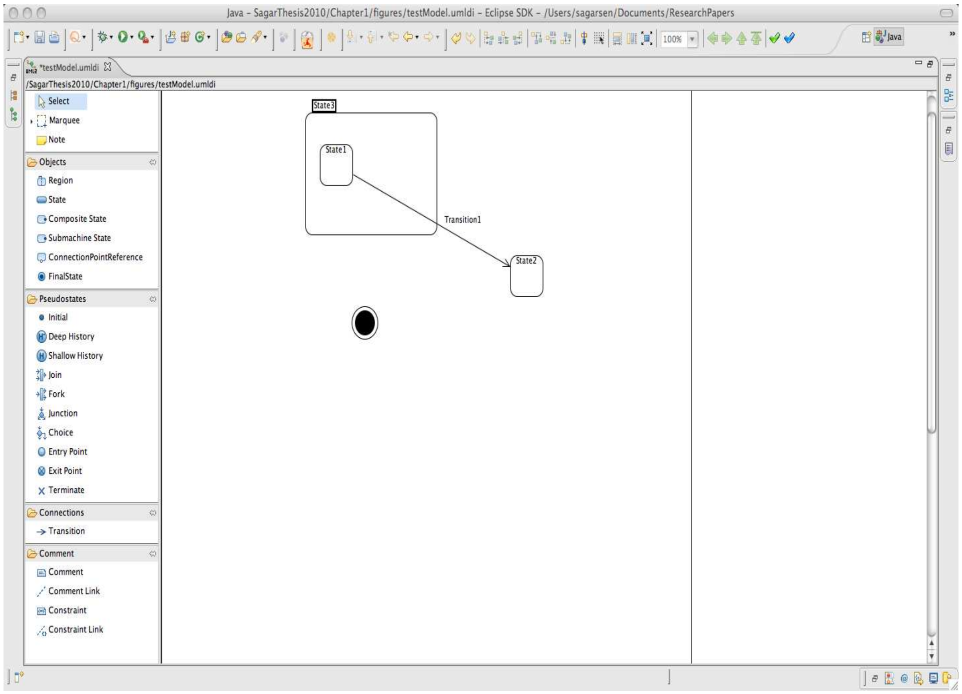


Figure 1.4: Partial Model in the TopCaseD UML Model Editor

valid UML state machine model and satisfies all the well-formedness rules of a state machine. What is probably more interesting is the nearest consistent UML state machine that contains all elements of the partial model. There may be a number of possibilities to complete such partially specified models. We can relate automatic model completion to the automatic code completion problem in programming environments [15]. This scenario raises the question: *How do we automatically discovery complete models or recommendations to complete partial models?*

1.3.3 Scenario 3: Generation of Products in a Software Product Line

A Software Product Line (SPL) references to a set of products sharing a common, managed set of features that satisfy the specific needs of a particular mission [37]. A *Feature Diagram* (FD) or a *feature model* specifies of a modelling domain for a SPL. Feature diagrams introduced by Kang et al. [77] [78] compactly represent all the products of an SPL in terms of features which can be composed. A FD consists of k features $f_1, f_2, ..., f_k$ and dependency constraints between features. For instance, selection of some features in a product may compulsorily link the selection of other features. Further, some of the features may be associated with a software asset such as web service. Consider the FD for a car crash crisis management system in Figure 1.5. The FD contains 47 features where 25 of them are optional. Some of the features are associated with services or software assets. The FD describes 335,54,432 different configurations of features. Can software assets in all configurations be composed into a valid product? Answering this

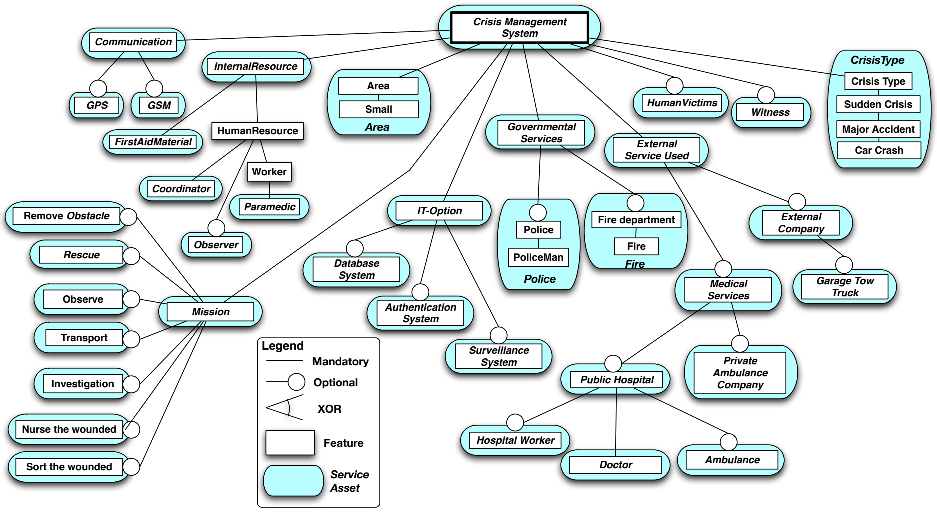


Figure 1.5: A Feature Diagram for Car Crash Crisis Management System

requires creating either all products or a representative subset of all products. For instance, what are the set of all products that contain all valid pairwise interaction between features? Creating these products will help us reveal invalid products. Manually creating products that satisfy all FD constraints is very tedious. Therefore, we ask, *how do we automate product generation in a software product line for various objectives?*

1.4 Problem Context and Challenges

We are motivated by the need for automatic generation of effective models in a modelling domain. The problem context for automatic model discovery is shown in Figure 1.6. The context identifes the following inputs:

- **Specification of a Modelling Domain:** The modelling domain specifies a set of models M. Examples of modelling domain specifications are metamodel for modelling languages and feature diagrams for SPLs.

- **Heterogenous Sources of Knowledge:** Heterogeneous sources of knowledge $Source_1, Source_2, ...Source_k$ possibly in different modelling languages specify subsets of the modelling domain $M_1, M_2, .., M_k$. The intersection of these subsets $M_1, M_2, ..., M_k$ is the effective modelling domain represented by a set of effective models $M_{effective}$. We can see the heterogeneous sources of knowledge as a set of constraints in different languages that limit the set of models M to a subset $M_{effective}$.

Given these inputs we ask: What is the automatic discovery mechanism that can create models in the set $M_{effective}$? This is the global question that intrigues us.

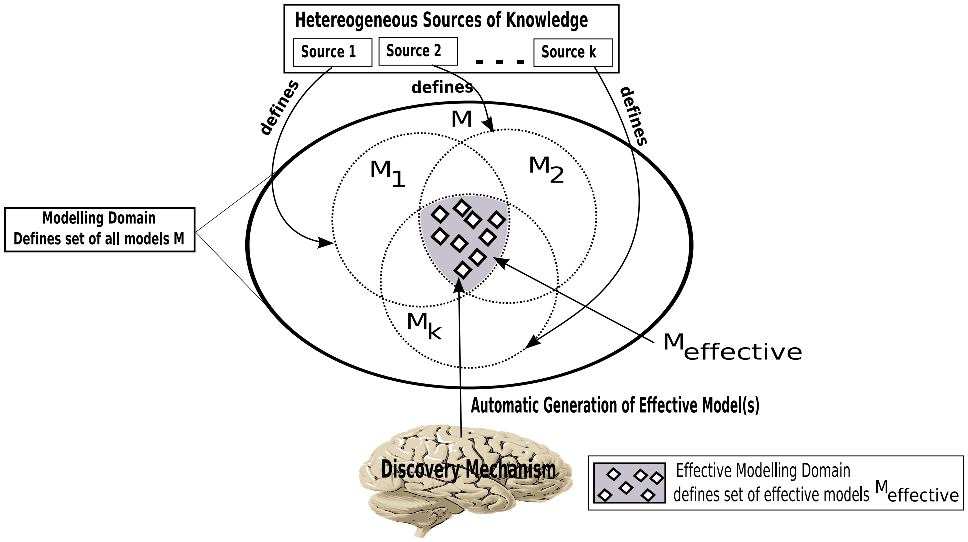

Hetereogeneous Sources of Knowledge
Source 1
Source 2
Source k
defines
defines
defines
M
M_1
M_2
M_k
Modelling Domain
Defines set of all models M
$M_{effective}$
Automatic Generation of Effective Model(s)
Discovery Mechanism
Effective Modelling Domain
defines set of effective models $M_{effective}$

Figure 1.6: Problem Context for Automatic Model Discovery

This question gives rise to a number of challenges pertaining to automatic model discovery. We describe the most important challenges below:

Challenge 1 Discovery mechanism: Generative vs. Constraint Satisfaction? Our research began with the exploration of existing mechanisms to automate the generation/discovery of models in a modelling domain. We classify existing approaches as either *generative* or those based on *constraint satisfaction*. The question was which approach is promising?

A generative approach attempts to incrementally create models in a modelling domain by object instantiation. For instance, in [29], the authors present an imperative algorithm and a tool to generate models that conform only to the Ecore specification of a metamodel. The approach does not ensure the satisfaction of constraints from heterogeneous sources of knowledge such as well-formedness rules. Similarly, in Ehrig et al. [52], the authors propose a graph grammar based approach to generate models that conform to a class diagram (or Ecore model). These models do not conform to any OCL constraints on the meta-model.

Constraint satisfaction based approaches attempts to transform a modelling domain to a set of variables and constraints on them. The set of constraints is solved using a constraint solver [91]. One or more low-level solutions are transformed as models of the modelling domain. This approach has been used in domain-specific settings such as software testing. The Korat (Chandra et al.) [28] system is able to generate data structures implemented in the Java Collections Framework that satisfy predicates. Similarly, Sarfraz Khurshid in his Ph.D. thesis [83] presents the TestEra tool for generating Java data structures such as linked lists, tree maps, hash sets, heap arrays, and binary trees for testing. Both approaches are limited to standard data structures and not to the more generic notion of models. The most intriguing approach was the tool UML2Alloy [92]. The tool attempts to transform UML class diagram models, that largely resemble metamodels, to the formal specification language ALLOY [72]. One may then use ALLOY to analyze UML models by generating examples and counterexamples. Although the tool is not directly related to model discovery it aims to transform class diagram constructs to a constraint satisfaction problem in ALLOY. However, UML2Alloy does not transform complex metamodel constructs such as multiple inheritance and multiple containments. UML2Alloy fails to solicit the use of ALLOY when the size of the UML model is large making the approach unscalable.

Generative approaches create models incrementally and cannot satisfy constraints simultaneously. Therefore, a number of models may need to be rejected as they may not satisfy constraints. Therefore, constraint satisfaction based approaches seem more promising.

Challenge 2. Transforming the specification of a modelling domain to a constraint satisfaction problem The specification of a modelling domain contains a set of concepts and relationships between them. These relationships might encode complex constraints that are not easily transformed to a constraint satisfaction problem. Further, a large number of concepts and relationships may lead to a very large constraint satisfaction problem that becomes computationally intractable.

For instance, the transformation of a metamodel specification to a constraint satisfaction problem requires a constraints model for constructs such as:

- Multiple Inheritance
- Multiple containers for a class
- Opposite properties
- Identity properties
- Composite properties

The large size of a metamodel such as the UML with about 246 classes hampers the direct transformation to a tractable constraint satisfaction problem.

Challenge 3. Transforming heterogeneous sources of knowledge to constraints Heterogeneous sources of knowledge are specified in different modelling languages. However, for constraint satisfaction they all need to be transformed to constraints in a common language. For instance, the task of generating test models for a model transformation must satisfy constraints specified in a textual constraint language such as Object Constraint Language, test objectives, and the pre-condition of the model transformation expressed in the language of the transformation.

Challenge 4. Generation of models must be within tractable and finite bounds The discovery of models in a modelling domain requires generation of models of finite size. What are the heuristics to determine the appropriate size of a model that is sufficient to satisfy knowledge from heterogenous sources of knowledge?

Challenge 5. Detection of Inconsistent Sources of Knowledge Knowledge from various sources may be inconsistent with respect to the modelling domain specification. How can we detect such inconsistent sources of knowledge and eliminate them?

Challenge 6. Validating the Effectiveness of Models There is a need to conduct *rigorous experiments* that qualify models generated by constraint satisfaction. The qualification guarantees whether models are effective or useful for given objectives. These experiments must consider the effect of various influencing factors on the quality of the generated models. For instance, one may ask what is the influence of generating multiple models using a particular constraint solver on their effectiveness as test models? Do different parameters to the constraint solvers drastically affect the quality of the solutions?

1.5 Thesis

In this thesis we propose that it is possible to automatically discovery effective models in a modelling domain. Categorically, we address the problem of effective model discovery in two modelling domains: (a) Metamodels (b) Feature Diagrams. A metamodel is a very general specification of a modelling language's domain. A metamodel can be used to specify the domain of any domain-specific modelling language. However, legacy software systems and components cannot always be modelled or remodeled in a modelling language from scratch. Ideally, time tested components must be reused in their legacy form for combination with other components to build a software system. If we see these legacy components as features then the possible combinations of features is best modelled using the feature diagram language giving rise to a Software Product Line. The coarse-grained components associated with features may be combined in different configurations which are part of the feature diagram modelling domain. This distinction between pure models in the domain of a modelling language and configurations of coarse-grained legacy components in a product line realize model-driven software construction at different levels .Therefore, we consider both specifications of modelling domains in this thesis.

Consequently, we propose two frameworks for model discovery specializing the general framework shown in Figure 1.6:

1. The framework for automatic effective model discovery in the modelling domain specified by a metamodel. This framework is embodied in the tool CARTIER.
2. The framework for automatic effective product discovery in the modelling domain specified by a feature diagram. This framework is embodied in the tool AVISHKAR.

1.5.1 A Framework for Automatic Effective Model Discovery

The Figure 1.7 presents the overall view of the framework for automatic effective model discovery. The framework is embodied in the tool CARTIER. The name CARTIER comes from the famous French discoverer from St. Malo who discovered Canadian in-lands in Quebec. The primary input to the framework is the specification of the modelling domain given by an *input metamodel*. The *input metamodel* MM_{in} specifies a set of models M. The input metamodel consists of a set of types (class with properties, enumeration, primitive) to instantiate models of a modelling language. Concretely, the input metamodel is stored as an instance of the ECORE metamodel which is part of the industry standard Eclipse Modeling Framework (EMF) [58]. The models themselves are stored as XMI [10] files representing instances of the Ecore metamodel.

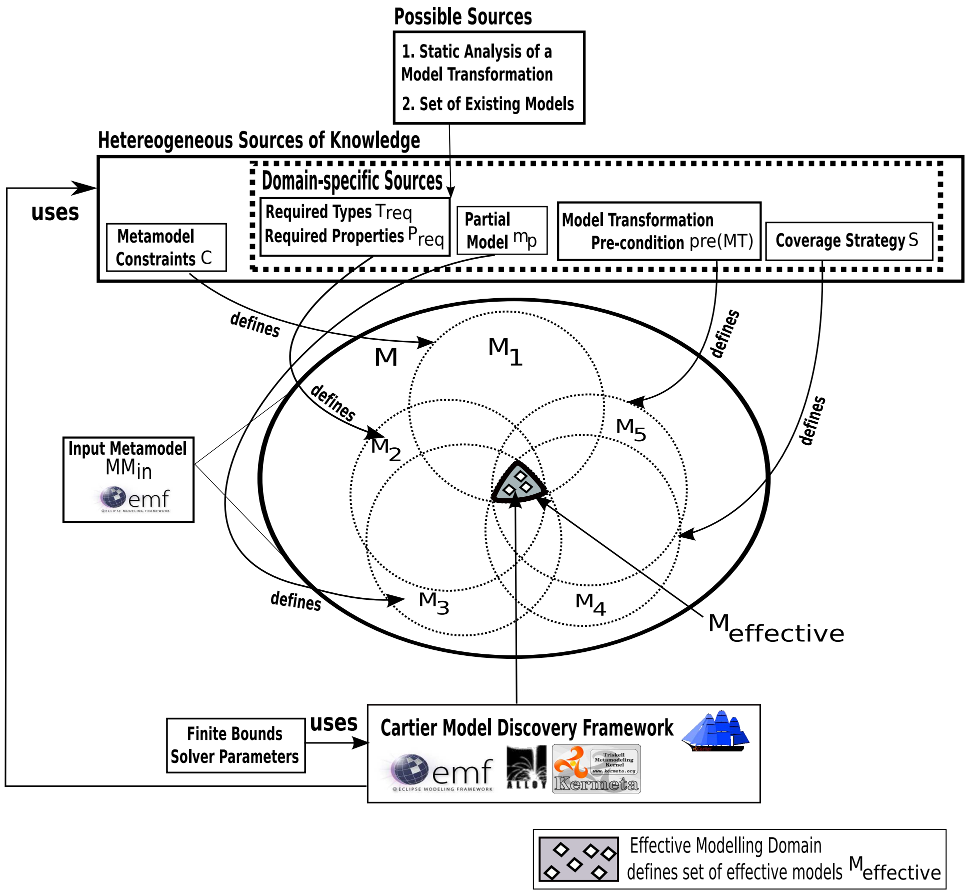

Possible Sources
1. Static Analysis of a Model Transformation
2. Set of Existing Models
Hetereogeneous Sources of Knowledge
uses
Domain-specific Sources
Required Types Treq
Required Properties Preq
Partial Model mp
Model Transformation Pre-condition pre(MT)
Coverage Strategy S
Metamodel Constraints C
defines
defines
defines
defines
defines
M
M1
M2
M3
M4
M5
Meffective
Input Metamodel MMin
emf
uses
Finite Bounds Solver Parameters
Cartier Model Discovery Framework
emf
ALLOY
Kermeta
Effective Modelling Domain
defines set of effective models Meffective

Figure 1.7: A Framework for Automatic Effective Model Discovery

Heterogenous sources of knowledge constrain the modelling domain specified by a metamodel:

- **Required types** T_{req} **and properties** P_{req} in the input metamodel. The set of required types and properties helps extract a subset of the input metamodel called the *effective metamodel*. The effective metamodel specifies the subset of models $M_1 \subset M$. There can be many possible sources for the set of required types and properties:
  - Static analysis of a model transformation gives a set of types and properties in the input metamodel actually manipulated by the transformation.
  - A set of models conforming to the input metamodel is another source of required types and properties. Visiting the models in the set gives us a set of types and properties used in the metamodel. A typical real-world example of this could be in a classroom setting for object-oriented design using UML. The professor can point out to students the required types and properties he used to create UML by visiting every object of a set of models automatically.
- **Metamodel Constraints** C are expressed on an input metamodel using a textual constraint language such as Object Constraint Language (OCL) [114]. These constraints encode restrictions that cannot be specified using a diagrammatic Ecore model. We illustrate this as the set $M_2 \subset M$.
- **Domain-specific sources of knowledge** may also help define the effective modelling domain. We present some of them below:
  - **Partial Model** m_p is a partially specified model using the input metamodel. For instance, a graphical model editor allows an user to create models in a modelling language such as UML state machines. An incomplete model in the editor is a partial model in the UML state machine language. The partial model may not respect all metamodel constraints of UML. Therefore, a partial model is often expressed as an instance of a *relaxed version of the input metamodel*. The partial model defines the subset $M_3 \subset M$.
  - **Coverage Strategy** S help define and generate *model fragments* [55] that cover a wide range of structural aspects in the input metamodel. For instance, an input domain partition based strategy helps generate a set of model fragments MF that cover partitions on all types and properties of the input metamodel. These model fragments help define an effective modelling domain for *coverage-based testing* of a model transformation. All test models that satisfy a coverage strategy contain the model fragments generated from the strategy. Model fragments are expressed in a modelling language that permits specification of ranges on properties of an input metamodel. A coverage strategy defines the subset $M_4 \subset M$.
  - **Transformation Pre-condition** $pre(MT)$ is a set of invariants on the metamodel that is specific to a model transformation MT. A model transformation often may not be designed to transform all models specified by its input metamodel. For instance, the

transformation from class diagram models to entity relationship diagram models [22] require that all classes in the input class diagram have at least one primary attribute. The OCL [114] is often used to express pre-conditions. A pre-condion defines the subset $M_5 \subset M$.

The intersection of all the sources of knowledge defines the *effective modelling domain*. The effective modelling domain is the set of models defined by $M_{effective} \leftarrow M \cap M_1 \cap M_2 \cap M_3 \cap M_4 \cap M_5$.

The methodology for model discovery uses the sources of knowledge presented above to automatically generate models in the effective modelling domain. We enlist the steps below:

Step 1. Effective Metamodel Identification : We prune the input metamodel MM_{in} to obtain the effective metamodel $MM_{effective}$ using a metamodel pruning algorithm [141]. The effective metamodel contains the set of required types T_{req} and properties P_{req} provided as input and all its obligatory dependencies computed by the metamodel pruning algorithm. All unnecessary types and properties are removed. $MM_{effective}$ is super type of MM_{in} from a type theoretic point of view and a subset of MM_{in} from a set-theoretic point of view. The size of the effective metamodel $MM_{effective}$ is often considerably smaller than the size of the input metamodel MM_{in}.

Step 2. Transformation of Effective Modelling Domain Specification to ALLOY : The effective modelling domain specification is defined by a number of artifacts. It is initially defined by the effective metamodel $MM_{effective}$ and constrained by knowledge from one or more sources: (b) Metamodel constraints C (b) Partial model m_p (c) Model fragments MF from coverage strategy S, and (d) Pre-condition $pre(MT)$ of a model transformation MT. We transform these artifacts expressed in possibly different languages to a **constraint satisfaction problem** (CSP) in the unique formal specification language ALLOY [71] [72]. The theoretical formalism for expressing the CSP is **first-order relational logic**.

Step 3. Generation of Models in Effective Modelling Domain : We solve the CSP in ALLOY to generate models in the effective modelling domain. CARTIER achieves this by invoking KodKod [53] in ALLOY to transform the CSP as relational model to Boolean Conjunctive Normal Form (CNF) . We invoke a satisfiability (SAT) solver such as MiniSAT [112], ZChaff [159] to solve the Boolean CNF. Finally, we transform low-level solutions of the CNF to models conforming to the input metamodel MM_{in}.

The generation of models in a modelling domain is often directed towards an objective. We need to ensure that the objective is consistently achieved considering all influencing factors. A typical question maybe what is the effect of a SAT solver on the quality of the solution? To answer this question we need to perform experiments that generate several solutions for the same constraint satisfaction problem. There are many other influencing factors for which we conduct rigorous experiments to validate discovery effectiveness. In this thesis, we perform experiments in the following application domains:

1. Test model generation for model transformation testing

2. Partial model completion in domain-specific model editors

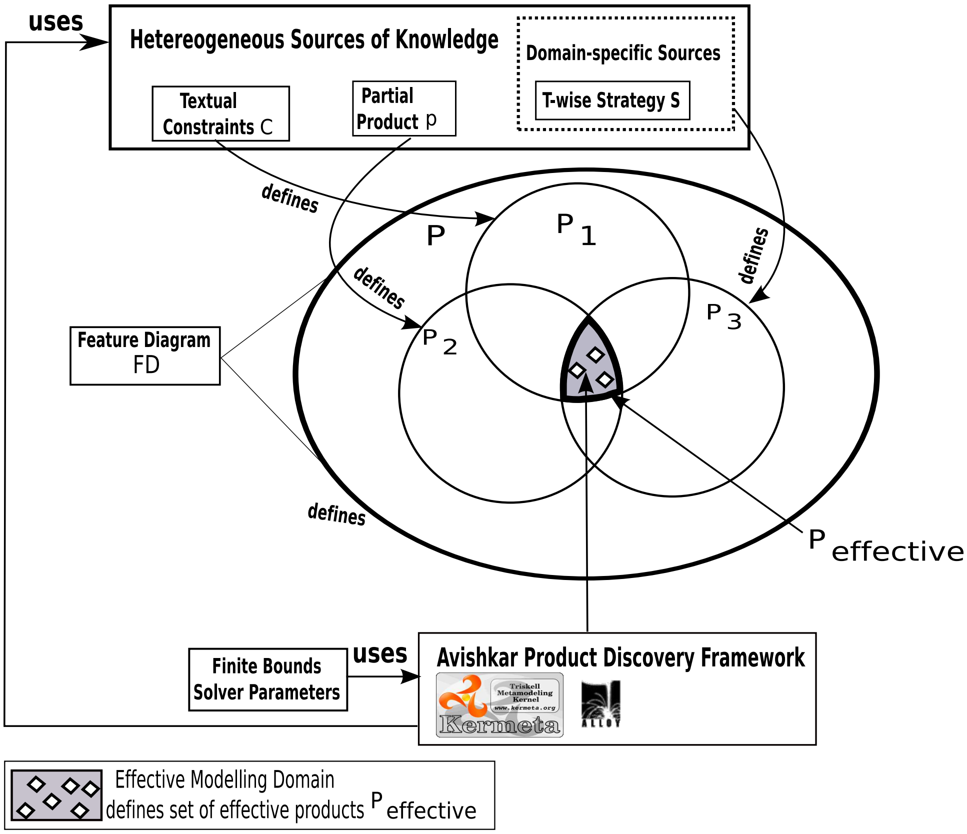

uses
Hetereogeneous Sources of Knowledge
Textual Constraints C
Partial Product p
Domain-specific Sources
T-wise Strategy S
defines
defines
defines
defines
P
P 1
P 2
P 3
Feature Diagram FD
P effective
Finite Bounds Solver Parameters
uses
Avishkar Product Discovery Framework
Triskell Metamodeling Kernel
www.kermeta.org
Kermeta
ALLOY
Effective Modelling Domain defines set of effective products P effective

Figure 1.8: A Framework for Automatic Product Discovery

1.5.2 A Framework for Automatic Effective Product Discovery

The Figure 1.8 presents the overall view of the effective product discovery framework. The framework is embodied in the tool AVISHKAR. AVISHKAR in Hindi means *Invention* which signifies the character of the tool to discover products in a SPL. The primary input to the framework is the specification of the modelling domain given by a *feature diagram* or *feature model*. The *feature diagram* FD specifies a set of products P. *Feature Diagrams* (FD) introduced by Kang et al. [78] compactly represent all the products (or configurations) of a SPL in terms of features which can be composed. Feature diagrams have been formalized to perform SPL analysis [136]. In [136], Schobbens et al. propose an generic formal definition of FD which subsumes many existing FD dialects. We define a FD as follows:

- A FD consists of k features $f_1, f_2, ..., f_k$
- A feature f_i may be associated with a software asset.
- Features are organized in a parent-child relationship in a tree T. A feature with no further children is called a leaf.
- A parent-child relationship between features f_p and f_c are categorized as follows:
  - *Mandatory* - child feature f_c is required if f_p is selected.
  - *Optional* - child feature f_c *may* be selected if f_p is selected.
  - *OR* - at least one of the child-features $f_{c1}, f_{c2}, .., f_{c3}$ of f_p must be selected.
  - *Alternative (XOR)* - one of the child-features $f_{c1}, f_{c2}, .., f_{ck}$ of f_p must be selected.
- Cross tree relationships between two features f_i and f_j in the tree T are categorized as follows:
  - f_i requires f_j - The selection of f_i in a product implies the selection of f_j.
  - f_i excludes f_j - f_i and f_j cannot be part of the same product and are *mutually exclusive*.

Using the FD we create products/configurations of features. We can compose software assets. associated with these features to derive the final product.

Heterogenous sources of knowledge constrain the modelling domain specified by a feature diagram:

- **Textual Constraints** C are expressed on a set of features. Constraints are expressed textually when they cannot be directly encoded in the FD. These constraints specify the subset $P_1 \subset P$
- **Partial Product** p is a set of features chosen in product. The set of features may require the selection of other features to derive a complete product. The partial product specifies the subset $P_2 \subset P$

- **T-wise Strategy** S is a product generation strategy to detect faults in software product lines [90] [120]. The large number of products specified by a feature diagram can be sampled using a strategy such as $T-wise$. The objective is to generate a minimum number of products that satisfy all $T-wise$ interactions between features. For instance, in a FD with 25 optional features (see Figure 1.5) specifies at least 2^{25} products. A $2-wise$ strategy where $T=2$ will lead to generation of only $4 \times {}_{25}C_2 = 300$ products that cover all pairwise interactions between features. The $T-wise$ strategy for a particular value of T specifies the subset $P_3 \subset P$.

The intersection of all the sources of knowledge defines the *effective modelling domain*. The effective modelling domain is the set of products defined by $P_{effective} \leftarrow P \cap P_1 \cap P_2 \cap P_3$.

The product discovery methodology uses the sources of knowledge presented above to automatically generate products in the effective modelling domain of a FD. We enlist the steps below:

Step 1. Transformation of Feature Diagram to ALLOY : We transform a feature diagram to constraint satisfaction problem in the formal specification language ALLOY [72] [71].

Optional Step. Transformation of Partial Product to ALLOY and their Completion : We can transform a partial product p to ALLOY. It generates an ALLOY predicate that represents the partial information about selected features in the partial product. It can then solve the ALLOY model to generate one or more complete products.

Step 2. Generation of $T-wise$ **Tuples and Detection of Valid Tuples using** ALLOY: In this thesis we focus on generating products that satisfy $T-wise$ interaction between features. We first generate ALLOY predicate represents $T-wise$ tuples and detects those that are not consistent with the constraints in the FD.

Step 3. Scalable Generation of Products We propose *divide-and-compose* strategies to generate a set of products that cover all valid tuples that cover $T-wise$ interactions between features. The approach splits the satisfaction problem for all tuples to solving subsets of tuples. We solve multiple ALLOY models with these subsets to obtain sets of products. The sets of products are merged into a final set of products.

Do products discovered using the framework consistently attain their objectives? For instance we may ask what is the effect of divide-and-compose strategies on the redundancy of products generated? To answer this question we need to generate products considering all important influencing factors. In this thesis, we validate our framework using rigourous experiments in the following application domains:

1. Test product generation that satisfy the t-wise interaction criteria

2. In ongoing/future work, we show that our framework can effectively sample the space of Quality of Service (QoS) of a dynamic web service who's variability is modelled as a FD.

1.6 Contributions

Both the frameworks for model and product discovery have led to the scientific contributions in this thesis. We explain these contributions in the following sub-sections. Some of the contributions are extracted and pin-pointed from the methodology already described in Section 1.5. We cite the relevant publications in peer-reviewed conferences and journals.

1.6.1 Contributions in Automatic Effective Model Discovery

Contribution 1.1 We present a comprehensive framework for generation of finite-sized effective models in any modelling language and constrained by heterogeneous sources of knowledge. The framework is embodied in the tool CARTIER. We use the formal specification language ALLOY for its ability represent constraints on graphs of objects and consequently to represent he entire metamodel as a constraint satisfaction problem. This contribution summarizes the answer to all challenges presented in Section 1.4 for a modelling domain specified by a metamodel. The tool CARTIER, saw its origins in our papers [130], [138].
Contribution 1.2. The framework transforms all metamodel constructs to ALLOY for constraint satisfaction. It also deals with metamodel with multiple inheritance by flattening it to single inheritance in ALLOY. Further, the framework presents transformation to ALLOY facts from constraints imposed by multiple containers, opposite properties, identify properties, and composite properties. This contribution addresses challenge 2 of Section 1.4. The transformation to ALLOY has been briefly described in two of our contributions [138] and [140].
Contribution 1.3. The framework is built using Kermeta modelling and model transformation language to simultaneously process models of knowledge in different languages. Each source of knowledge is expressed as a model in a modelling language. For instance, model fragments are expressed as models of a model fragment language. Kermeta can load, save, and manipulate models conforming to different metamodels at the same time. Therefore, CARTIER, written in Kermeta, transforms knowledge from various models to facts in the target language ALLOY. This contribution addresses challenge 3 of Section 1.4 and is published in our papers [138] [103].
Contribution 1.4. In the framework we present a metamodel pruning algorithm [141] that uses a set of required types and properties to generate an effective metamodel from large input metamodel. The effective metamodel is often very small and can be easily transformed to ALLOY as a tractable constraint satisfaction problem. This contribution addresses part of challenge 2 of Section 1.4 and presented in the paper [141].
Contribution 1.5. The framework contains facilities to assign finite bounds to the number of objects for each type in the model. It also transforms the solutions from the SAT solver in ALLOY called ALLOY instances back to high-level model conforming to a metamodel. The generation of models conforming to heterogeneous sources of knowledge helps determine inconsistencies between them if any. A selection of inconsistent sources of knowledge is made and either modified or eliminated from the specification of the effective modelling domain. This contribution addresses challenges 4 and 5 of Section 1.4 and is published in articles [138] and [140].
Contribution 1.6. We validate models generated for their effectiveness using the framework by performing the following experiments:

- **Test model generation for model transformation testing :** We generate thousands of models for a representative transformation. We use mutation analysis [107] to demonstrate that test models generated using *partitioning strategy* can detect 93% of the bugs compared to arbitrary generation 70%. We show that the partitioning strategy is not affected by various biases such as dependence on the ALLOY solver. The experimental study is published in [139] and journal version of the paper [128] has been submitted.

- **Partial model completion in domain-specific model editors:** We use our framework to generate recommendations to complete partial models in the model editor AToM^3 [67]. We illustrate that our framework can automatically complete partial models in a model editor. The experiments show that this can be done for small examples within reasonable time limits. This work is published in [131], [140].

This contribution addresses challenge 6 of Section 1.4.

1.6.2 Contributions in Automatic Effective Product Discovery

Contribution 2.1. We present a comprehensive framework for generation of effective products in a Software Product Line specified by a feature diagram. The framework is embodied in the tool AVISHKAR. The framework contains the transformation of a feature diagram to a constraint satisfaction problem in ALLOY. The framework invokes a solver on the ALLOY model to automatically generate products conforming to the feature diagram. This contribution summarizes the answer to all challenges in Section 1.4 for a modelling domain specified by a feature diagram.
Contribution 2.2. Given a set of feature selections (available/not available) the framework uses ALLOY to detect if a product can be created such that these feature selections are satisfied. A constraint for instance states that features f_1 exists in the product, while f_2 should not exist. If f_2 is a mandatory feature then AVISHKAR uses ALLOY to detect that the constraint is invalid. This contribution addresses challenge 5 of Section 1.4.
Contribution 2.3. Scalable generation of test products from a feature diagram Feature diagrams have been transformed to constraint satisfaction problems for testing a software product line. For instance, Cohen et. al. have applied combinatorial interaction testing to systematically select configurations/products [42] from a feature diagram. They consider various algorithms in order to compute configurations that satisfy pair-wise and t-wise criteria [41]. The constraints imposed due to feature relationships in a feature model are solved by calling SAT solvers such as ZChaff [159]. However, their approach is not very scalable when we consider large feature diagrams. Our framework contains *divide-and-compose* strategies to split the problem of test product generation satisfying $T-wise$ into sub-problems. The tool AVISHKAR solves the sub-problems and merges the results into a small set of products that contain all valid tuples required by the $T-wise$ criteria. This mechanism renders our methodology to be a scalable approach to generate products in a software product line. This contribution addresses challenge 4 of Section 1.4.
Contribution 2.4. Validation of Effectiveness of Test Products: There is a need to perform experiments that qualify the products generated using our framework. We perform experiments to generate products for a transaction processing feature diagram AspectOPTIMA. We show that *redundancy* in $T-wise$ tuples is introduced in the products due to divide-and-compose strategies. In on-going work we perform experiments to generate different configurations of a dynamic web-service orchestration. We demonstrate that the QoS of a web-service varies with different configurations of the web-service. These variable QoS analysis experiments help us define an effective methodology to set robust contractual agreements for dynamic web service.

The above contributions are published in [120]. Two papers [14] [80] have been submitted to apply the product discovery tool AVISHKAR to analysis of varying QoS in a web service orchestration.

1.7 Thesis Organization

The thesis contains 6 chapters including the introduction. The next 5 chapters are organized as follows:

- Chapter 2, we introduce the context of MDE and the state of the art in automatic effective model discovery in a modelling domain.
- Chapter 3, we present automatic effective model discovery in the domain specified by a metamodel.
- Chapter 4, presents empirical validation of the framework for model discovery. In particular, we focus on two application domains for validation: (a) test model generation for a model transformation (b) partial model completion in the model editor AToM^3
- Chapter 5, we describe the framework for automatic test product discovery in a software product line. We empirically validate the framework for the redundancy in the generated products.
- Chapter 6, we summarize our work and present perspectives for future research. We briefly describe ongoing work on analysis of variable QoS in a dynamic web service.

Chapter 2

Context and State of the Art

This chapter describes the context and state of art for automatic discovery of effective models in a modelling domain. In Section 2.1, we describe *Model Driven Engineering* (MDE) which provides the philosophy and tools to specify modelling domains and transformations between them. We describe the creation or specification of two modelling domains (a) *Metamodels* for modelling languages in Section 2.2 (b) *Feature diagrams* for products in a Software Product Line in Section 2.3. Models in a modelling domain are transformed using the model transformation language Kermeta to the formal specification language ALLOY. In Section 2.4, we present Kermeta and its important features such as extensibility using *aspects* and *model typing*. In Section 2.5, we describe the formal specification language ALLOY.

After describing the context and technological foundations needed for this thesis we present the state of the art in the proposed scientific contributions. In Section 2.6, we present the state of the art in various aspects of automatic discovery for the modelling domain specified by a metamodel. In Section 2.6.1, we present related work on identifying an effective modelling domain. In this thesis we perform mode discovery experiments in test model generation and partial model completion in model editors. We present the related work for test model generation in Section 2.6.3 and partial model completion in model editors in Section 2.6.4.

In Section 2.7, we present the state of the art in test product discovery in the modelling domain specified by a feature diagram for SPLs. We perform product discovery experiments in analyzing the variability in QoS of dynamic web services.

2.1 Model-driven Engineering

MDE [110] is a philosophy and a set of tools to help simplify and accelerate complex software development. The simplification in development is achieved by exalting the creation of software from the level of programs to first class artifacts called *models*. Models are graphs of inter-connected objects in a *modelling domain*. Different models in a domain are created using a common set of domain-specific/problem-specific concepts and relationships. For instance, the well-known general purpose modelling language UML [116] is used to create various high-level models of software design using concepts in UML class diagrams. These UML models contain only objects of UML concepts/types. UML models are at a higher level of abstraction with respect to code in a general-purpose programming language such as Java where use of the language pervades all aspects of software development. MDE prescribes that a domain expert should find it easier to reason in his problem domain using models instead of directly writing code. Are models simply data structures or can they be transformed, evolved, or executed? The MDE answer to this question is a *model transformation*. Model transformations help transform high-level or domain-specific models to other models or executable code in a language such as Java. The automation offered by model transformations such as a code generator ultimately helps accelerate software development.

We set ourselves the specific goal of automatic discovery of models in a modelling domain. This goal solicits answers to two important questions in MDE:

1. How to specify a modelling domain and create models in it?

2. How do we transform models from one modelling domain to another?

The **first question** is addressed in this paragraph. The specification of a modelling domain consists of a set of concepts, relationships between concepts, and some invariants on the structural relationship between objects. For instance, a *metamodel* specifies the modelling domain of all models in a modelling language. For instance, the UML metamodel specifies infinite UML models. Metamodels can be created in the EMF *Ecore* format to specify the modelling domain of a modelling language. Similarly, the modelling domain of all products in a SPL is specified by a *feature diagram*. Models in a modelling domain can be instantiated by (a) Creating objects of concepts specified in a modelling domain specification (b) Assigning properties to these objects to build relationships. The models must also satisfy a set of invariants on their structure. The Object Constraint Language (OCL) is often used to specify structural invariants on models in a modelling domain. The EMF provides the set of software tools to specify modelling domains, create models within these domains, and validate these models against invariants. Detailed description of the modelling domain for metamodels is given in Section 2.2 while in Section 2.3 we present the specification for feature diagrams.

The **second question** is addressed in this paragraph. Once, we create the specification of a modelling domain and models within them we see the need to transform these models. Models can be transformed within the same modelling domain or between modelling domains. Model transformation [142] languages such as the imperative Kermeta [82] [108], rule-based ATL [76] [75] [3], graph grammar based AToM^3 [67], Viatra [156] transform models. Model transformation languages are expected to conform to the Query-View-Transformation (QVT) standard [75]. Different types of model transformations can be created using these languages as classified in [44]. Model transformations may transform models within the same language (endogenous transformations), between different languages (exogenous transformations) and even realize the classical view of generating executable code from a high-level model. In this thesis, we use the Kermeta model transformation language which we describe in Section 2.4.

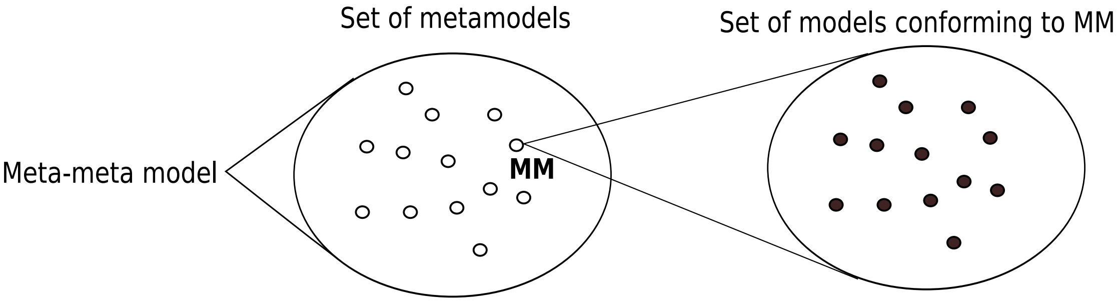


Figure 2.1: Set-theoretic View of a Modelling Domain specified by a Metamodel

2.2 Metamodel Specification of a Modelling Domain

In Figure 2.1, we present a set-theoretic view of the modelling domain specified by a metamodel. The metamodel *MM* specifies a possibly infinite set of models in a *modelling language*. The metamodel *MM* itself is a model in the set of all metamodels. The set of *all metamodels* is specified by a *meta-meta modelling language*. The meta-meta modelling language allows the specification of concepts and relationships between them. Historically, the Entity-Relationship diagram (ER Diagram) [36] has been one of the most popular meta-meta level languages used to specify *database schemas* for databases in various domains. In MDE, the Class Diagram and its dialects [9] are widely used to specify a metamodel. The EMF standardized the ECORE modelling language to specify metamodels. A natural question is how can one specify the meta-meta modelling language? The answer is that meta-meta modelling languages are expressive enough to specify themselves. For instance, in Figure 2.2 we present the metamodel for ECORE in ECORE itself. This property of a meta-metamodelling language is known as *boot strapping*. The metamodel for ECORE is a model in the set of all metamodels. We do not go into the details of describing the ECORE metamodel which is given detail in [58]. We illustrate the specification of a metamodel using ECORE in the following section.

2.2.1 Specification of a Metamodel

The Ecore metamodel in Figure 2.2 presents the various concepts one can use to specify metamodels. Most notably, instances of classes *EClass, EReference, EAttribute, EEnum, EOperation*, and *EParameter* are used to specify metamodels. For convenience, we remove the prefix *E* and use the familiar names class, property (for reference or attribute), enumeration, operation, and parameter in the text. We describe the specification of a simple language to represent Hierarchical Finite State Machine (HFSM) using ECORE. The metamodel for HFSM is shown in Figure 2.3. One possible sequence of steps to specify a metamodel is the following:

1. **Specification of Class and Enumeration Types**: Classes and enumerations in a metamodel are created. For instance, we create the classes HFSM, Transition, AbstractState, and State. One may do this concretely using either the ECORE tree editor available in EMF or using an ECORE diagram editor available with tools such as TopCASED [54].

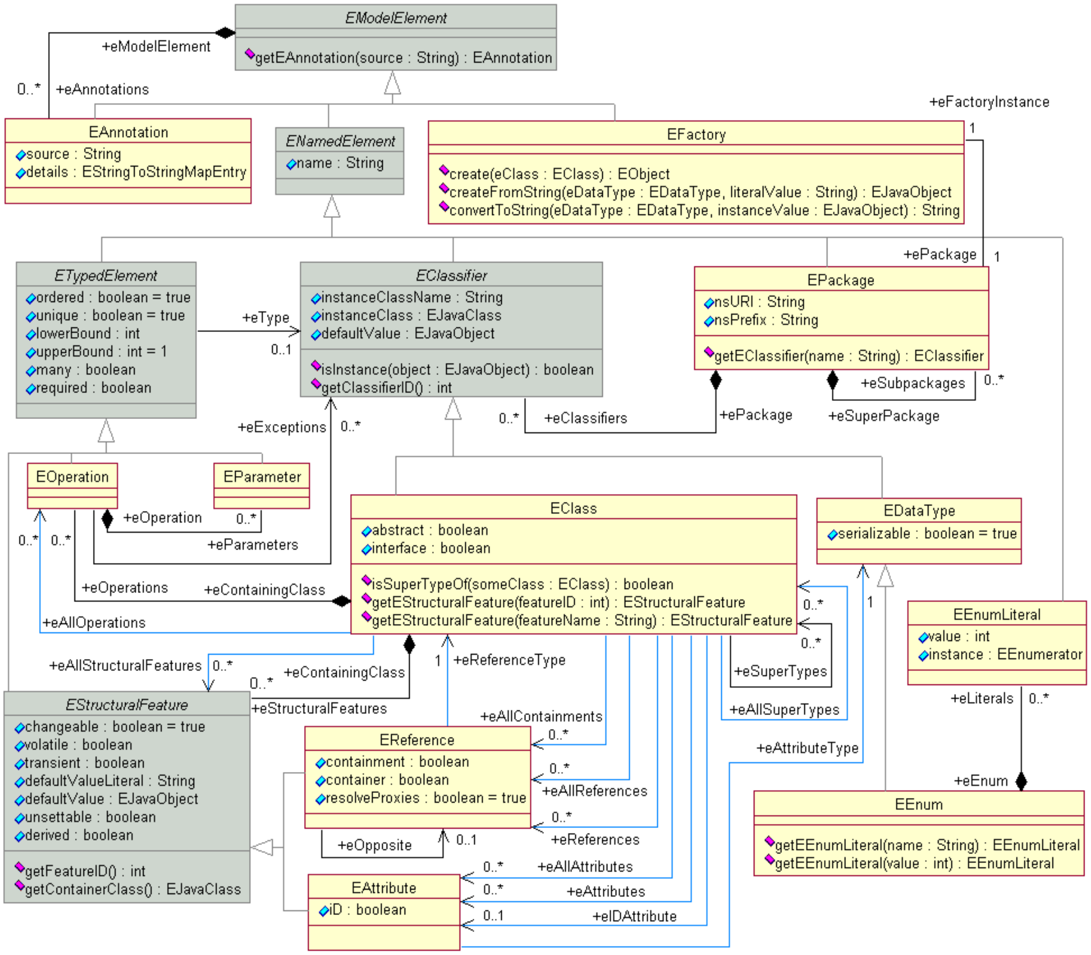


Figure 2.2: ECORE Metamodel

2. **Specification of Class Hierarchy**: Some classes inherit references and attributes from other classes. For instance, we create the inheritance hierarchy for classes State and Composite that inherit from the class AbstractState. ECORE allows specification of multiple and multilevel inheritance where a class can inherit reference from several classes.

3. **Specification of Properties**: Properties which include references and attributes are inserted into classes. For instance, the event property in Transition is a primitive attribute of String type. Similarly, the property incomingTransition of class AbstractState is a reference of type Transition. An ECORE editor can be used to insert attributes into a class and create references from a class to other classes.

4. **Specializing Properties**: There are several ways to add more meaning to a property. Some of the important characteristics of a property are:

   - **Composite Property:** A composite property of type Class B owned by a Class A implies that A is a possible container for objects of class B. If an object of class B is contained in Class A then it cannot be contained by other classes. For instance, the composite property HFSM.states indicated by the black diamond implies that all objects of type AbstractState are contained in exactly one HFSM object.
   - **Opposite or Bi-directional Property:** The opposite or bi-directional property bind two objects using the same relationship. For instance, Transition.target and AbstractState.incomingTransition are opposite properties. Any object of type Transition that refers to a target State object will enforce that the target State object has an incoming Transition object.
   - **Multiplicity of a Property:** A property can have variable multiplicity or cardinality indicating the size of an attribute or the number of references. For instance, the property Composite.ownedState has the multiplicity 0..*.

5. **Specification of Operations**: Operations are included in a class to specify the operational or denotational semantics for a model or a part of it. For instance, the operation HFSM.run() executes the HFSM. An operation may be code in a general purpose language such as Java a high-level state chart model, or a model of computation.

2.2.2 Object Constraint Language to Specify Metamodel Constraints

The specification of a metamodel is a starting point to describe concepts and their relationships in a modelling language. It also includes some implicit constraints such as inheritance, specialization of properties. However, a metamodel is still limited in its use to specify constraints on the content and structure of models in a modelling language. Some constraints are better expressed in a textual constraint language, We specify constraints on a metamodel using the Object Constraint Language (OCL) [114]. The OCL is an Object Management Group(OMG) standard to specify side-effect free constraints on models conforming to a metamodel. The entire OCL specification is available in [114].

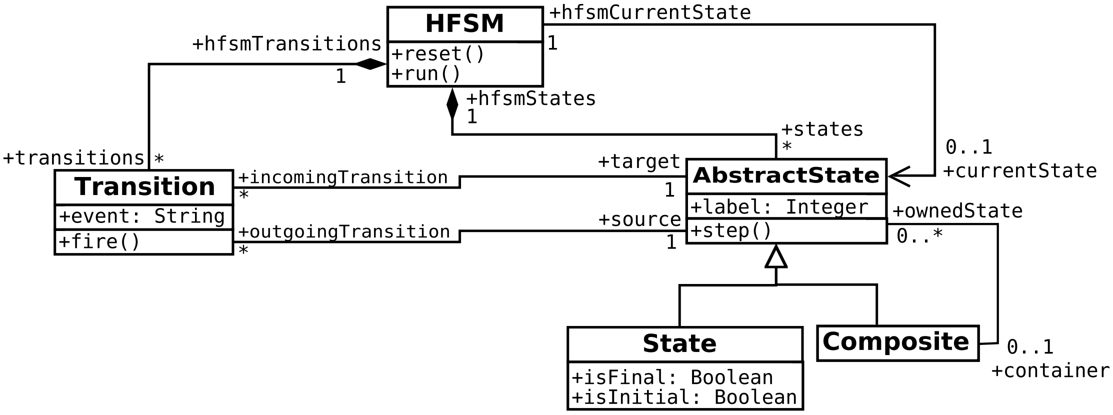


Figure 2.3: Hierarchical Finite State Machine Metamodel

We may specify constraints on the HFSM modelling language in OCL. For instance, the constraint that *there must be only one initial state* in a HFSM model is expressed in OCL as:-

$context\ State\ inv:$
$State.allInstances() \rightarrow select(s|s.isInitial = True) \rightarrow size() = 1$

Dissecting the OCL constraint we observe that a constraint is specified within a *context*. In this constraint the context is the class State. The constraints first creates a temporary subset, say I, of the set of of all objects/instances of the State class. The subset I contains State object with the property isInitial set to $True$. Further, the constraint states that the size of the subset must be equal to one. Overall, the constraint checks if the model contains exactly one initial State object. This constraint is *side effect free* which means it does not enforce any property on the model.

In general, OCL language statements are constructed in four parts:

1. A *context* that defines the limited situation in which the statement is valid
2. A *property* that represents some characteristics of the context (e.g., if the context is a class, a property might be an attribute)
3. An *operation* (e.g., arithmetic, set-oriented) that manipulates or qualifies a property, and
4. *Keywords* (e.g., if, then, else, and, or, not, implies) that are used to specify conditional expressions.

OCL is also used a navigation language for models that conform to a metamodel.

2.2.3 Models in the Modelling Domain

The metamodel specification of a modelling domain allow the instantiation or creation of models in it. Using ECORE one may create instances of classes in a metamodel.

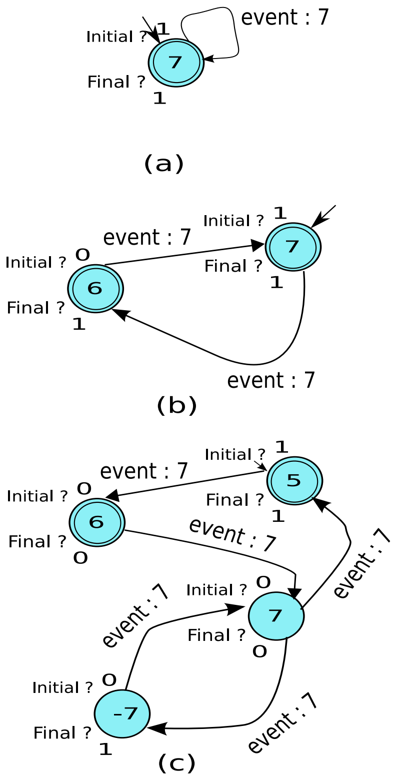


Figure 2.4: Examples of HFSM models

Some examples of valid models in the HFSM modelling language are shown in Figure 2.4. The models are shown in their concrete syntax. All models are created using objects of the HFSM metamodel and satisfy OCL constraints on the HFSM metamodel. For instance, all models satisfy the constraint that there must a path from any state to a final state, all models have exactly one initial state and at least one final state.

2.3 Feature Diagram Specification of a Modelling Domain

In Figure 2.5, we present the set-theoretic view of the modelling domain specified by a *feature diagram*. A feature diagram FD specifies a set of *products* in a Software Product Line. For instance, software on different Nokia phones are different instances of the same product line of mobile software adapted to different hardware configurations.

The feature diagram itself is a model in the set of all possible feature diagrams. The set of all feature diagrams is specified using the *feature diagram modelling language*. The feature diagram modelling language allows creation of a feature diagram containing various product line features and their inter-dependencies. The feature diagram modelling language is specified using a metamodel. We describe this metamodel in Section 2.3.1. We describe the creation

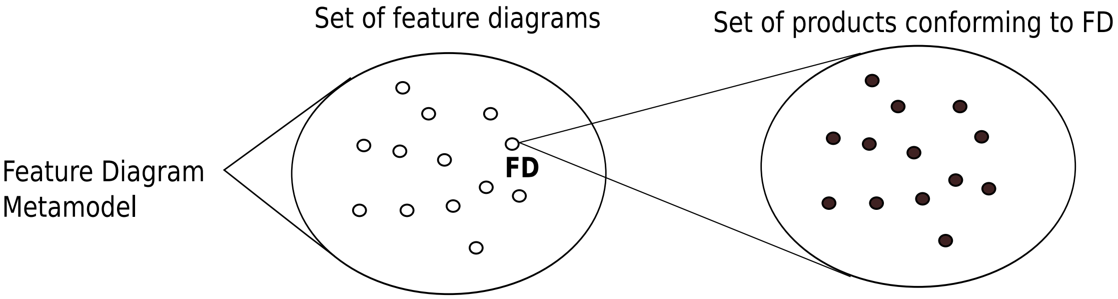


Figure 2.5: The Modelling Domain of a Feature Diagram

of a feature diagram as an instance of this metamodel in Section 2.3.2. In Section 2.3.3, we demonstrate how products are instantiated from the feature diagram.

2.3.1 The Feature Diagram Modelling Language

Variability being at the heart of the software product line appraoch, the community came up with several ways fo documenting SPL variability either in the form of UML profiles [162, 63] or domain specific languages [154, 77]. In particular, Feature Diagrams [1] are widespread due to their simplicity and conciseness. However, since their original definition, a plethora of feature modeling notations have been proposed ([43, 61, 78] to name a few). Indeed, feature models can be considered as a product line of notations sharing commonalities and exposing differences which are not always explicitly defined.

In such a context, there is a risk of being dependent of a particular feature modeling notation both raising the issue of its selection and unnecessarily restricts the applicability of our approach. Fortunately, Schobbens et al. [136, 134] performed a formal analysis of the existing feature modeling notations. To do so, they developed a pivot abstract syntax called Free Feature Diagrams (FFDs) used to map any feature modeling construct found in existing notations in order to reason formally on the syntax and semantics of these notations. The universal nature of FFDs makes it suitable for various applications; we used it to reason on variability [62] and to support product derivation in a model-driven way [119]. In order to process feature models, we derived in [119] an EMF metamodel from FFD's abstract syntax. We recall this formalization here since it will serve as the main foundation to specify our coverage strategies as well as quality metrics of the generated configurations.

FFDs are defined in terms of a parametric structure whose parameters serve to characterize each FD notation variant. GT (Graph Type) is a boolean parameter indicating whether the considered notation is a Direct Acyclic Graph (DAG) or a tree. NT (Node Type) is the set of boolean operators available for this FD notation. These operators are of the form op_k with $k \in \mathbb{N}$ denoting the number of children nodes on which they apply to. Considered operators are and_k (mandatory nodes), xor_k (alternative nodes) or_k (true if any of its child nodes is selected), opt_k (optional nodes). Finally $vp(i..j)_k$ ($i \in \mathbb{N}$ and $j \in \mathbb{N} \cup *$) is true if at least i and at most j of its k nodes are selected. Existing other boolean operators can usually be expressed with

[1] we also use the term "Feature Models" interchangeably with "Feature Diagrams"

vp. The union of $vp(i..j)_k$ is called $card$. GCT (Graphical Constraint Type) is the set of binary boolean functions that can be expressed graphically. A typical example is the “requires” between two features. Finally, TCL (Textual Constraint Language) tells if and how boolean constraints defined over the set of FD nodes can be defined. With the help of these sets, a generic abstract syntax for FDs is given. A FD is then composed of the following elements:

- A set of nodes N, which is further decomposed into a set of primitive nodes P (which have a direct interest for the product). Other nodes are used for decomposition purposes. A special root node, r represents the top of the decomposition,
- A function $\lambda : N \mapsto NT$ that labels each node with a boolean operator,
- A set $DE \in N \times N$ of decomposition edges. As FDs are directed, node $n1, n2 \in N, (n1, n2) \in DE$ will be noted $n1 \rightarrow n2$ where n1 is the *parent* and n2 the *child*,
- A set $CE \in N \times GCT \times N$ of constraint edges,
- A set $\phi \in TCL$

A FD has also some well-formedness rules to be valid: only root (r) has no parent; a FD is acyclic; if GT = true the graph is a tree; the arity of boolean operators must be respected.

These constructs were used to build an ECORE based metamodel depicted in Figure 2.6. The metamodel is proposed in the paper [119] . Its constitution was driven by simplicity and pragmatism. *FeatureDiagram* is the root class of the metamodel. This class has an attribute $graphTypeTree$ corresponding to the boolean GT (Graph Type) presented previously. It also contains a list of features (class Feature) corresponding to the set of nodes N . The special root node r is identified by the reference root from *FeatureDiagram* to *Feature*. The authors of [119] keep all base operators (because they are simple and widely used) rather than using exclusively card like operators. In the metamodel, these operators are subtype of the abstract class Operator, and each feature (class Feature) contains 0 or 1 operator (that corresponds to the function?). The class Feature also contains a list of edges (class Edge) allowing the construction of the set DE of decomposition edges. The set CE of constraint edges is represented in the metamodel by the class $ConstraintEdge$ and they are contained by the class FeatureDiagram. Each *ConstraintEdge* contains either a *Require* constraint or a $Mutex$ constraint. Primary feature nodes are related to UML models (see below) defining the core assets involved in the realization of these features. In the metamodel, a primary feature is related to UML models by the composite association between the class Feature and the class Model. Finally, well-formedness rules (Feature Modeling Constraints) have been implemented in terms of constraints boolean constraints on the FD.

2.3.2 Specification of a Feature Diagram

The feature modelling language described in the previous section can be used to create an FD representing a Software Product Line. For instance, we present the AspectOPTIMA FD in Figure 2.7. The FD for AspectOPTIMA contains 19 features allowing maximum of 2^{19} configurations when FD constraints are neglected.

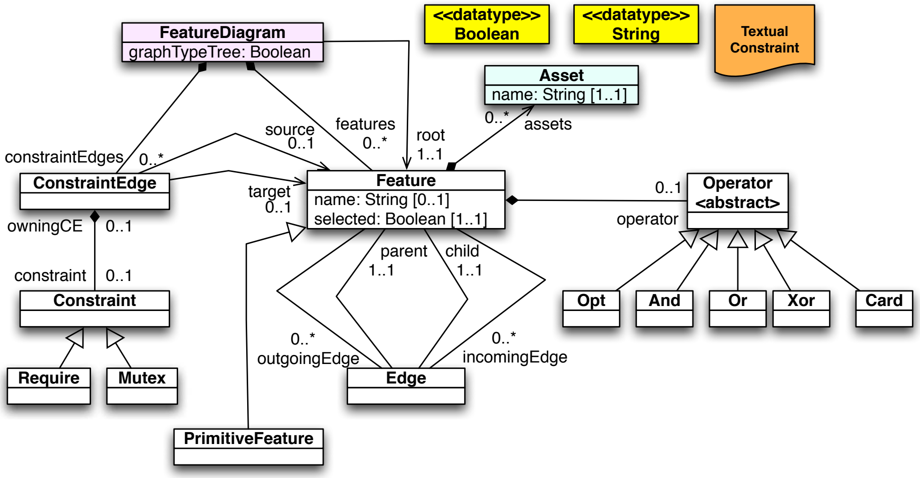


Figure 2.6: The Feature Diagram Metamodel

2.3.3 Products in the Modelling Domain of a Feature Diagram

A feature diagram models the domain of a finite number of products. A *Product* corresponds to a selection of features in the *FD* such that it satisfies all restrictions in the *FD*. For instance, we present three different products in Figure 2.8 for the AspectOPTIMA *FD* in Figure 2.7.

2.4 Modelling and Model Transformation Language: Kermeta

In this thesis, we use Kermeta as the common language to both represent modelling domains and to express transformations between them. This section briefly describes Kermeta and some of the its important features used in the implementation of CARTIER and AVISHKAR.

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [115]. The object-oriented meta-language MOF supports the definition of metamodels in terms of object-oriented structures (packages, classes, properties, and operations). It also provides model-specific constructions such as containments and associations between classes. Kermeta extends the MOF with an imperative action language for specifying constraints and operational semantics for metamodels [108]. Kermeta is built on top of EMF within the ECLIPSE development environment. The action language of Kermeta provides mechanisms for dynamic binding, reflection, and exception handling. It also includes classical control structures such as blocks, conditionals, and loops.

2.4.1 Aspect-weaving in Kermeta

The first key feature of Kermeta is its ability to extend an existing metamodel with constraints, new structural elements (meta-classes, classes, properties, and operations), and functionalities

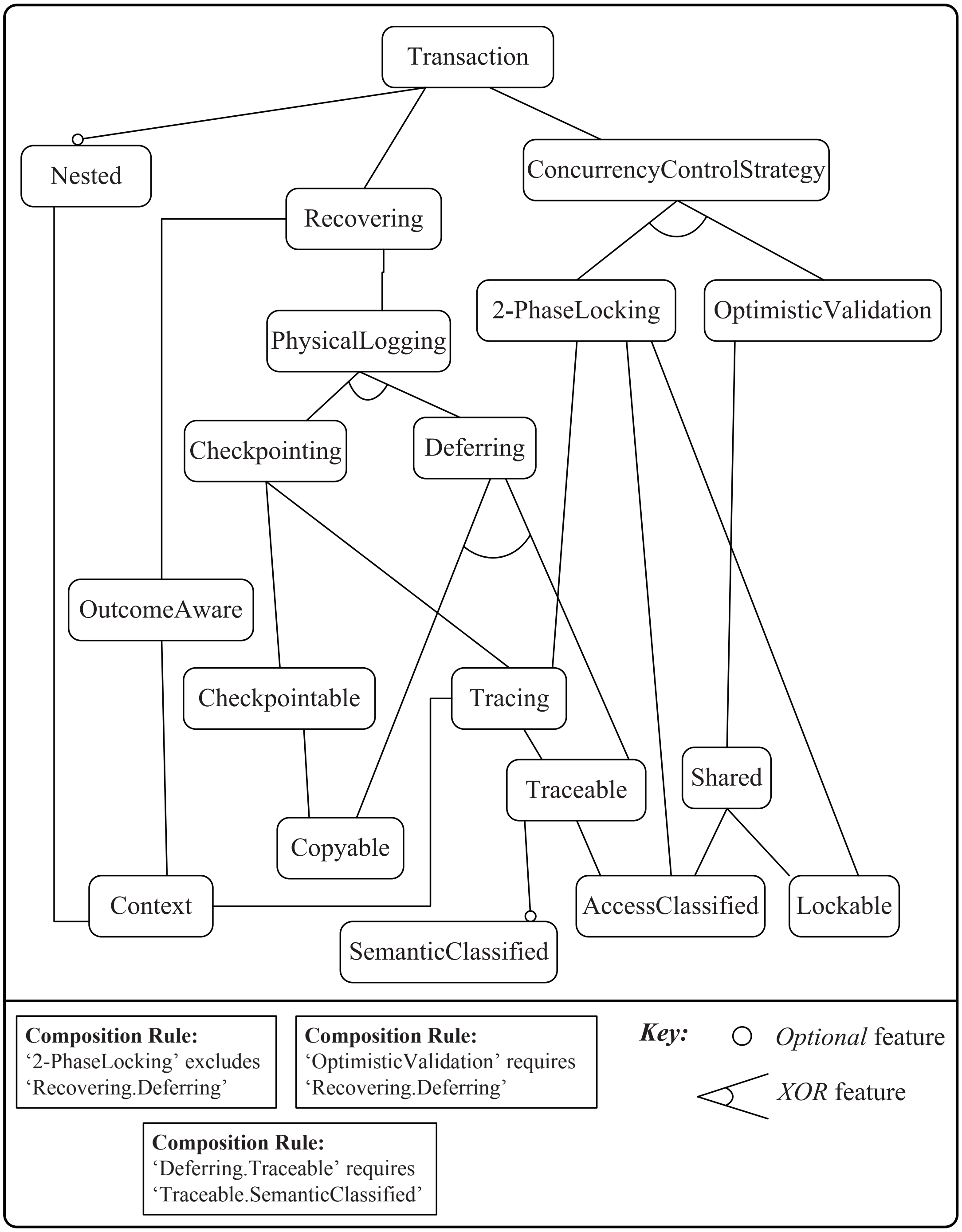

Transaction
Nested
Recovering
ConcurrencyControlStrategy
2-PhaseLocking
OptimisticValidation
PhysicalLogging
Checkpointing
Deferring
OutcomeAware
Checkpointable
Tracing
Traceable
Shared
Copyable
Context
AccessClassified
Lockable
SemanticClassified
Composition Rule: '2-PhaseLocking' excludes 'Recovering.Deferring'
Composition Rule: 'OptimisticValidation' requires 'Recovering.Deferring'
Composition Rule: 'Deferring.Traceable' requires 'Traceable.SemanticClassified'
Key:
Optional feature
XOR feature

Figure 2.7: The AspectOptima Feature Diagram

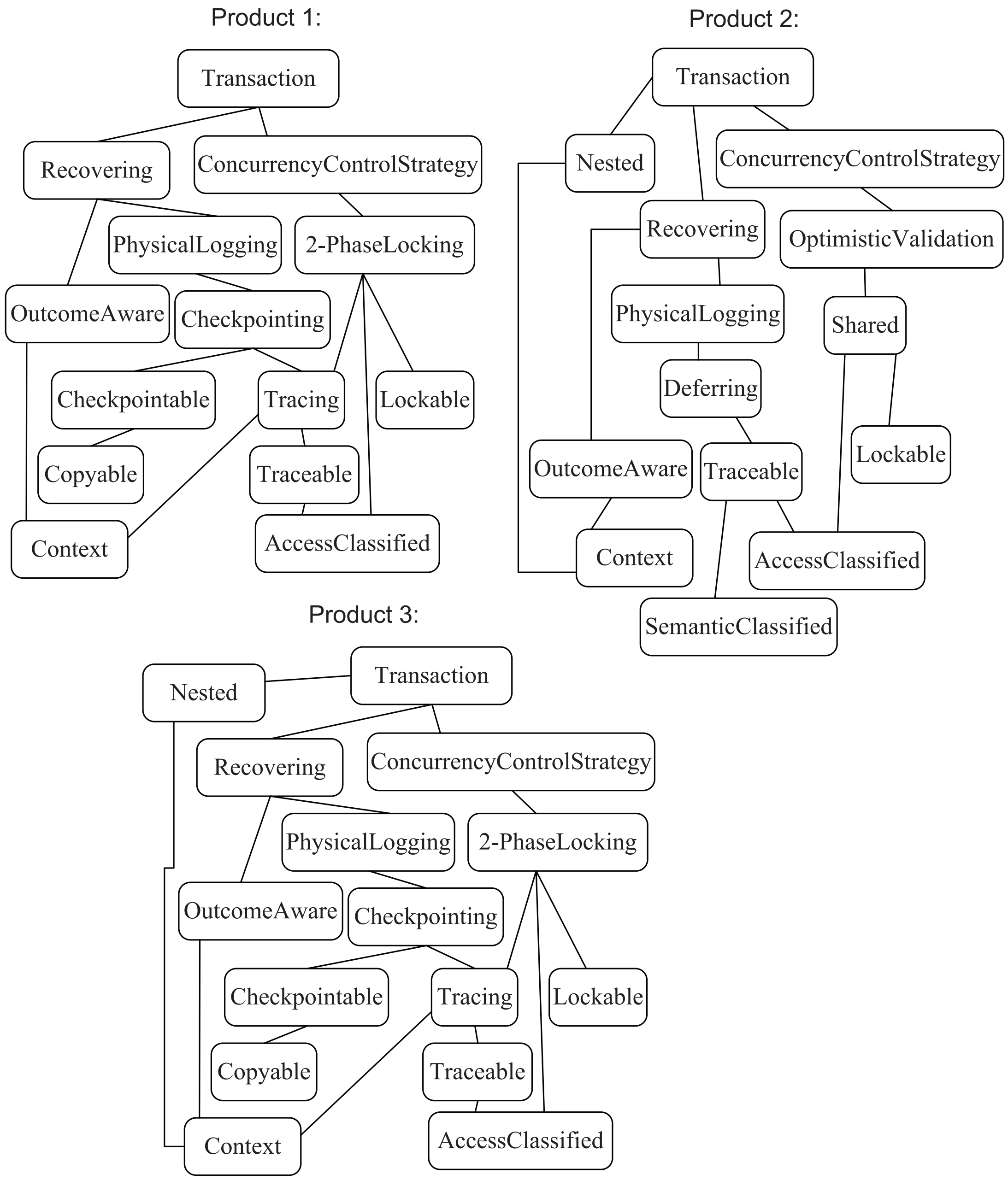


Figure 2.8: Three Products from the AspectOPTIMA feature diagram

defined with other languages using the *aspect* keyword. This keyword permits the composition of corresponding code within the underlying metamodel as if it were a native element of the metamodel. This feature offers more flexibility to developers by enabling them to easily manipulate and reuse existing metamodels.

The static composition operator "*require*" allows defining various aspects in separate units and integrating them automatically into the metamodel. The composition is performed statically and the composed metamodel is type-checked to ensure the safe integration of all units. This mechanism can be compared to the *open class paradigm* [38].

Open classes in Kermeta are used to organize "cross-cutting" concerns separately from their metamodel, a key feature of aspect-oriented programming [84]. Thanks to this composition operator, Kermeta remains a kernel platform and safely integrates all concerns around a metamodel.

Kermeta offers expressions very similar to Object Constraint Language (OCL) expressions [114]. In particular, Kermeta includes lexical closures similar to OCL iterators on collections such as each, collect, select, or detect.

Moreover, Kermeta also allows the direct importation and evaluation of OCL constraints. Pre-conditions and post-conditions can be defined for operations and invariants on classes.

Kermeta and its framework remain dedicated to model processing but provide an easy integration with other languages. Kermeta also allows importing Java classes to use services such as file input/output or network communications, which are not available in the Kermeta framework. It is also very useful, for instance, to make models communicate with existing Java applications.

In this thesis, we have made considerable use of aspect-weaving to weave properties and operations into metamodels with the goal of creating model transformations between modelling domains. For instance, we weave a reference to an input metamodel element into the output metamodel. Consequently, we weave an operation into the output metamodel that helps create an output model element using information from this reference. This direct referencing due to aspect-weaving eliminates the need to create intermediate data structures such as dynamic hash tables commonly used in compilers.

2.4.2 Model Typing with Kermeta

In Kermeta metamodels are also model types from a type-theoretic point of view. In this thesis, we solicit the use of model typing to check type conformance between metamodels before and after a transformation.

Model typing corresponds to a simple extension to object-oriented typing in a model-oriented context [146]. A model typing is a strategy for typing models as collections of interconnected objects. Model typing permits the detection of type errors early in the design process of model transformation. Moreover, it allows more flexible reuse of model transformations across various metamodels, while preserving type safety [146]. Type safety is guaranteed by type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type groups and type group matching [30]. The matching relation, denoted $<\#$, between two metamodels defines a function of the set of classes they contain according to the following definition:

> Metamodel *M'* matches another metamodel *M* (denoted *M' <# M*) iff for each class *C* in *M*, there is one and only one corresponding class *C'* in *M'* such that every property *p* and operation *op* in *M.C* matches in *M'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *M.C*.

This definition is adapted from [146] and improved here by relaxing the constraint related to the name-dependent conformance on properties and operations.

Let's illustrate model typing with two metamodels *M* and *M'* given in Figures 2.9 and 2.10. These two metamodels have properties and references that have different names. The metamodel *M'* has additional elements compared to the metamodel *M*.

> *C1 <# COne* because for each property *COne.p* of type *D* (namely, *COne.name* and *COne.aCTwo*), there is a matching property *C1.q* of type *D'* (namely, *C1.id* and *C1.aC2*), such that *D' <# D*.
>
> Thus, *C1 <# COne* requires *D' <# D*:
>
> - *COne.name* and *C1.id* are both of type *String*.
> - *COne.aCTwo* is of type *CTwo* and *C1.aC2* is of type *C2*, so *C1 <# COne* requires *C2 <# CTwo*. And, *C2 <# CTwo* is true because *CTwo.element* and *C2.elem* are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [145]. The interested reader can find the details of matching rules used for model types in [145].

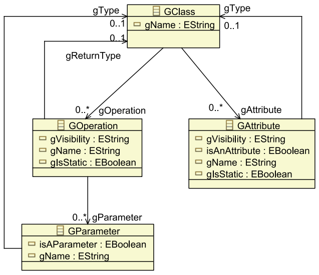


Figure 2.9: Metamodel *M*.

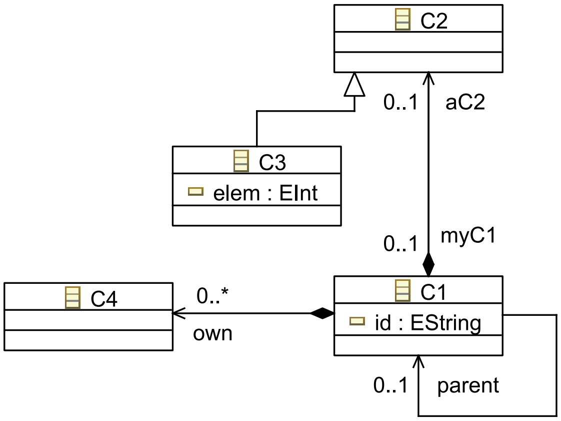


Figure 2.10: Metamodel *M'*.

2.5 Formal Specification Language: Alloy

In this thesis, we transform the specification a modelling domain and heterogeneous sources of knowledge to *constraint satisfaction problem*. The constraint satisfaction problem is expressed in the formal specification language: Alloy [72] [71].

Alloy is a structural modelling language based on **first-order relational logic**. Alloy was originally conceived by Daniel Jackson and developed by the Software Design Group at MIT. Alloy is conceived to specify and analyze the conceptual design of an object-oriented software system. Analysis includes generation of instances of a design to detect for example abnormal and generating counterexamples for assertions on the design. The analysis helps detect design flaws.

In this thesis, we use Alloy as a target language to specify a modelling domain and heterogeneous sources of knowledge as a *constraint satisfaction problem* (CSP). An Alloy *instance* or *solution* is a model that satisfies the CSP. We obtain these instances by solving the Alloy model in a *finite scope* The scope of an instance is the limit on its size. Generation of instance of models in Alloy is based on the hypothesis that finite and small models are useful in most real-world applications.

A CSP in Alloy model consists of the following important *paragraphs*:

```
module HFSM
open util/boolean as Bool
//Alloy Signatures
one sig HFSM
{
  states:set AbstractState,
  currentState: lone AbstractState,
  transitions: set Transition
}

abstract sig AbstractState
{
  label: Int,
  outgoingTransition: set Transition,
  incomingTransition: set Transition,
  container: lone Composite,
  hfsmCurrentState: one HFSM,
  hfsmStates: one HFSM
}

sig Transition
{
  event: Int,
  target: one AbstractState,
  source: one AbstractState,
  hfsmTransitions:one HFSM
}

sig State extends AbstractState
{
  isFinal: one Bool,
  isInitial: one Bool
}

sig Composite extends AbstractState
{
  ownedStates: set AbstractState
}
```

Listing 2.1: Signatures for HFSM metamodel

```
//Example Alloy Facts

//The HFSM must contain exactly one initial state
fact exactlyOneInitialState
{
  one s:State|s.isInitial == True
}
//The HFSM must contain at least one final state
fact atleastOneFinalState
{
  some s:State | s.isFinal == True
}
//There is exactly one HFSM object
fact exactlyOneHFSM
{
  one HFSM
}
//All AbstractStates have unique labels
fact AbstractState_label_unique
{
  all s1:AbstractState ,s2:AbstractState | s1!=s2=>s1.label != s2.label
}

// A Composite State Cannot Contain Itself
fact compositeCannotContainItself
{
  all c1:Composite, c2:Composite | c1  = c2 => c2 not in c1.ownedStates and c1 not in c2.ownedStates
}
```

Listing 2.2: Facts for HFSM metamodel

```
//All Composite States in the Model must contain at least 2 owned States
pred ExamplePredicate
{
 all c:Composite | #c.ownedStates > 2
}
```

Listing 2.3: An Example Predicate

```
//Example 1
run ExamplePredicate for 10

//Example 2
run ExamplePredicate for exactly 3 State, exactly 1 Composite, 1 HFSM, 5 Transition
```

Listing 2.4: Example Run Commands

- **Signatures and Fields:** A *signature* is used to model a concept or a class of objects in ALLOY. A signature contains *fields* that represent properties of concepts. For instance, we may model the classes in the HFSM metamodel (see Figure 2.3) as ALLOY signatures with fields as shown in Listing 2.1. A signature can be an abstract signature such as AbstracState. Only objects or instances of signatures are present a solution to an ALLOY model. A field in a signature can have a multiplicity one, lone (0 or 1), or it can be a set. It also has a type which refers to a primitive signature such as Integer or another signature in the ALLOY model. For instance, the field incomingTransition in the signature AbstractState is a set of signature type Transition. The field isFinal of signature State has a multiplicity one and is of type Bool. The signature Bool for Boolean is defined in another module imported using an open declaration.

- **Facts:** Facts are constraints on signatures and fields in the declarative ALLOY model. A fact must always hold true. For instance, we may express some facts on the HFSM metamodel as shown in the Listing 2.2. A fact often contains expressions that specify a constraint on sets of objects using quantifiers such as *all* (∀), *some* (∃), *one*, and *none*. For instance, the fact Abstract_label_unique states that for any two states s1 and s2, if s1 is not s2 then their labels are different hence enforcing the unique label constraint.

- **Predicates:** Predicates in ALLOY are constraints that need not always hold true like facts. They may be satisfied by selection by the modeller with the signatures and the facts. Predicates may be used to model knowledge from various sources as constraints in the ALLOY model. For instance, the predicate in Listing 2.3 states that all composite states in the HFSM must contain at least 2 states. The predicate is not a fact that must be true for all HFSM models but a constraint that represents a specific objective or requirement.

- **Run Command:** We may try to satisfy a predicate in an ALLOY model by attempting to generate instances in a finite scope. The ALLOY run command is used describe the finite scope of the solution size. For instance, the first example in Listing 2.4, attempts to obtain an HFSM instance up to a scope of 10. This implies that every there may be a maximum of 10 instances for each signature. The second example in Listing 2.4 presents qualifiers for the scope of each signature. For instance, the qualifier *exactly* 3 State enforces all instances to contain exactly 3 States.

An ALLOY model is transformed to a relational model in the relational model finder KodKod [53]. At relational level of abstraction the model structure is comprised of primitive entities called *atoms* and *relations* that define the relationship between atoms. All signatures represent the set of atoms. All fields, facts, and predicates represent relations between atoms.

An atom is a primitive entity that is:

- *Indivisible*: It can't be broken down into smaller parts

- *Immutable*: Its properties don't change over time; and

- *Uninterpreted*: It doesn't have any built-in properties, the way numbers do for instance.

A relation is a set of tuples where each tuple is a sequence of atoms. ALLOY is based on first-order logic and hence relations cannot contain other relations. The number of atoms in a relation is its *arity*. A relation can be unary, binary, ternary or can contain more atoms. A relation with three or more atoms is called a *multi-relation*. For instance, the ternary relation State =$\{(State0),(State2),(State3)\}$ represents 3 State atoms.
In ALLOY logic the basic entity is a *relation*. Even an atom is represented as a singleton set in relation tuple.

Relations represent the structure of graphs of objects in MDE. For instance, the isInitial property in the HFSM metamodel (see Figure 2.3) may be modelled as a relation
isInitial=$\{(State0,False),(State1,True),(State2,False)\}$.

Restrictions or constraints on structure in ALLOY is expressed as disallowed relations between atoms. ALLOY provides several operators to express constraints on relations including set operators, logical operators, and most notably relational operators such as the dot operator (for navigating structure), quantification operators (to specify constraints on a set of atoms), and multiplicity constraints.

To obtain a solution to the relational model the ALLOY specification is transformed using KodKod [53] to a Boolean Conjunctive Normal Form (CNF) formula. The resulting satisfaction problem is solved using a Boolean Satisfiability (SAT) solver such as MiniSAT [112] or ZChaff [159].

2.6 State of the Art in Model Discovery in a Modelling Language

Automatic model discovery in a modelling domain has many components. Previous work has proposed problems and solutions to one or more of these components. In this section we present related work for the following components of automatic model discovery:

- **Effective Modelling Domain Identification**
- **Generation of Models in a Modelling Domain**

Further, we present the state of the art in validating automatic model discovery for two application domains:

- **Experiments in Test Model Generation**
- **Experiments in Partial Model Completion in a Model Editor**

2.6.1 Related Work for Effective Modelling Domain Identification

There has always been a need to define the effective modelling domain for a given objective in MDE. This is true especially in the case of using large General Purpose Modelling Languages (GPMLs) such as UML. In this section we present related work that deal with the problem of obtaining and using the effective modelling domain.

Consider a fundamental task in MDE: Creating a model in a model editor such as in the Eclipse [58] environment. A popular editor for UML models is TOPCASED [54]. The tool can be used to create UML models such as class diagrams, state machines, activity diagrams, and use-case diagrams. If a modeller chooses to create class diagrams the tool presents the user with modelling elements for class diagrams such as classes and associations but not UML state machine modelling elements such as states and transitions. Therefore, the tool inherently prevents the modeller from using an unnecessary part of the UML meta-model. The *hard-coded* user interface in TOPCASED in fact presents the modeller with an effective modelling domain.

Model transformations on GPMLs such as UML are built for specific tasks and can process only a sub-domain of its huge input domain. To filter the input to a model transformation *pre-conditions* [142] are specified in a constraint language such as Object Constraint Language (OCL) [114] [93]. Graph transformation based model transformation languages specify pre-conditions to apply a graph rewriting rule on a left-hand side model pattern [147].

In the paper [144] Solberg et al. present the issue of navigating the meta-muddle notably the UML meta-model. They propose the development of Query/Extraction tools that allow developers to query the metamodel and to extract specified views from the metamodel. These tools should be capable of extracting simple derived relationships between concepts and more complex views that consist of derived relationships among many concepts. They mention the need to extract such views for different applications such as to define the domain of a model transformation and extracting a smaller metamodel from the concepts used in a model. Meta-modelling tools such as those developed by Xactium [96] and Adaptive Software [1] possess some of these abilities. The authors of [144] propose the use of *aspects* to extract such views. However, the authors do not elaborate on the objectives behind generating such views.

In this thesis, we present a technique called metamodel pruning [8] [141] that extracts the effective metamodel from an input metamodel. The effective metamodel contains on the required classes and properties and their obligatory dependencies.

2.6.2 Related Work for Generation of Models in a Modelling Domain

We classify generation of models in a modelling domain as (a) Generation by construction (b) Generation by solving constraints.

Approaches for generation by construction aim to create correct models by incrementally constructing them. We review two such approaches. In Brottier et. al. [29], the authors attempt to incrementally generate models conforming to a metamodel using model fragments. However, a number of the models are rejected as they do not satisfy constraints on the metamodel. A very similar approach in [52] makes use of graph grammar rules to incrementally construct models. This approach for generating instances also suffers from the same problem of not being able to satisfy metamodel constraints.

Approaches for generation by constraint satisfaction aim to generate whole models that satisfy constraints *all at once*. In [130], the authors present a transformation for a partial model to a constraint satisfaction problem in PROLOG. The metamodel used to express the partial model is also transformed to a set of PROLOG constraints. The authors use PROLOG to automatically complete the partial model. However, PROLOG does not allow expression of constraints on sets of objects. Therefore, there is always a need for a partial model that defines the exact number of objects in the model. The metamodel constraints are transformed to low-level PROLOG constraints on the variables in the model. In [70], transform UML models that are very similar to metamodels to PROLOG for verification. Both approaches use PROLOG which lacks the ability to specify constraints on set of objects.

In this thesis, we preset CARTIER that transforms a metamodel to ALLOY. Transformation of a meta-model specification from UML to ALLOY has previously been explored in the tool UML2ALLOY [26] [92] [25]. UML2ALLOY supports transformation from meta-model concepts to ALLOY model concepts such as class to signature, property to signature field, operation to function, enumeration/enumeration literal to extends signature, and constraints to predicates. In our approach to transforming a meta-model to an ALLOY model we keep the same transformation format such we transform classes to signatures and properties to class fields. In UML2ALLOY composition and aggregation are transformed first to OCL constraints and then to ALLOY. In our tool we transform composition and aggregation in a meta-model directly to ALLOY facts. Our, approach to transforming single inheritance is the same as in UML2ALLOY. Inheritance is transformed to an ALLOY signature that extends an other ALLOY signature. We use CARTIER to also transform metamodels with multiple inheritance to ALLOY which is not addressed by UML2ALLOY. There is no clear specification in UML2ALLOY related articles [26] [92] [25] about transforming multiplicities to ALLOY. In our case we transform multiplicity constraints to ALLOY signature fields in case of occurrence of 0, 0..1, or 0..$*$ multiplicities. If the multiplicity is variable such as $a..b$ we synthesize an ALLOY fact constraining the size of a set of relations. The constraints in meta-model is restricted to a small subset of OCL as UML2ALLOY transforms only this subset of OCL to ALLOY. However, in CARTIER we propose the user to directly enter ALLOY predicates and facts in the ALLOY model giving the user the flexibility of expressing a wider range of constraints (those that have not been implemented in UML2ALLOY) such constraints with transitive closure which cannot be expressed directly in OCL. We also present a method to synthesize ALLOY predicates from a partial model. This use of partial knowledge to synthesize complete models greatly reduces model development time. The tool UML2ALLOY, does not support the use of partial model knowledge to help generate models.

2.6.3 Related Work for Test Model Generation

The first application of automatic model discovery is test model generation for model transformation. We explore three main areas of related work : test criteria, automatic test generation, and qualification of strategies.

The first area we explore is work on test criteria in the context of model transformations in MDE. Random generation and input domain partitioning based test criteria are two widely studied and compared strategies in software engineering (non MDE) [153] [158] [64]. To extend such test criteria to MDE we have presented in [55] input domain partitioning of input meta-models in the form of model fragments. However, there exists no experimental or theoretical study to qualify the approach proposed in [55].

Experimental qualification of the test strategies require techniques for automatic model generation. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as dealt with in the Korat tool of Chandra et al. [28]. Korat is faster than ALLOY in generating data structures such as binary trees, lists, and heap arrays from the Java Collections Framework but it does not consider the general case of models which are arbitrarily constrained graphs of objects. The constraints on models makes model generation a different problem than generating test suites for context-free grammar-based software [98] which do not contain domain-specific constraints.

Test models are complex graphs that must conform to an input meta-model specification, a transformation pre-condition and additional knowledge such as model fragments to help detect bugs. As cited earlier, in [29] the authors present an automated generation technique for models that conform only to the class diagram of a meta-model specification. A similar methodology using graph transformation rules is presented in [52]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [130], we present a method to generate models given partial models by transforming the meta-model and partial model to a Constraint Logic Programming (CLP). We solve the resulting CLP to give model(s) that conform to the input domain. However, the approach does not add new objects to the model. We assume that the number and types of models in the partial model is sufficient for obtaining complete models. The constraints in this system are limited to first-order horn clause logic.

The qualification of a set of test models can be based on several criteria such as code and rule coverage for white box testing, satisfaction of post-condition or mutation analysis for black/grey box testing. In this thesis, we are interested in obtaining the relative adequacy of a test set using mutation analysis [49]. In previous work [107] we extend mutation analysis to MDE by developing mutation operators for model transformation languages.

In this thesis, we use CARTIER for automatic test model generation. CARTIER transforms the input metamodel, pre-condition of a model transformation and test strategies to a constraint satisfaction problem in ALLOY. We solve the ALLOY model to generate test cases or models for the transformation. We use the mutation analysis technique for model transformations proposed in [107] to validate the effectiveness of the test cases in bug detection.

2.6.4 Related Work for Completion in Editors

The second application of automatic model discovery is partial model completion in a model editor. We explore existing language-directed editors that aim to use the specification of a language domain or modelling domain to complete partial code or models.

Language-directed editors have been around for since the early 1980s. Some of the well-cited research on language-directed editors are Mentor [50], Interlisp [149], Program Synthesizer [148], Rational [16], PECAN [125], and Gandalf [65]. Most of the existing language-based editors such as in Eclipse are based on *attribute grammars*[59]. These systems have been widely adopted and integrated in many editors for tasks such as syntax highlighting and syntax-directed editing. The *openArchitectureWare* [7] framework , based on the Ecore [58] meta-modelling framework, supports automatic sentence completion already implemented in Eclipse to help make recommendations to sentences in textual domain-specific modelling languages. These suggestions for sentence completion are based on the textual syntax of the modelling language and do not consider the complete consistency of the model with respect to the meta-model and constraints of the language.

In *Model Driven Engineering* (MDE), models built in domain-specific model editors pose a new challenge. The challenge is to complete a partial model specified in the model editor. This involves the editor to use domain-specific modelling language constraints to direct the completion of the partial model. Simply put, this involves constraint solving using knowledge described in the partial model to synthesize a model that conforms to the domain-specific modelling language. Constraint solving for model synthesis has been well-studied in the literature such as model design space exploration [132], partial model completion using Prolog [130] and constraint logic programming [89]. In [131], the authors present a model completion system in a domain-specific editor by combining knowledge from the meta-model and the partial model specified in the model editor to SWI-Prolog. The Prolog program is solved using a backtracking based solver to return results to the domain-specific environment which was originally synthesized by AToM^3 using the meta-model. The methodology is valid for any domain-specific modelling language in the limits of first-order Horn clause logic of SWI-Prolog. However, their primary limitation is that the number of objects in the complete model is equal to the number of objects in the partial model. No new objects are suggested by the model completion system and the user is limited to specifying only the correct number of objects in the partial model. This is primarily due to the fact that constraints are specified at the object property level in SWI-Prolog and not at the meta-level such as on sets of objects.

We identify the need to develop a model completion system that can automatically suggest complete models especially for DSML meta-models containing constraints both on sets of objects and their properties. This typically involves mapping of a meta-model and constraints based DSML specification to a mathematical formalism with tool support that solves constraints to give correct instances of the DSML. Notably, such instances should contain the network of objects (original object identities need not be preserved) specified in the partial model and additional objects (if required) with appropriate property values such that the complete model conforms to its DSML. We would also like to control the maximum size or scope of the complete model for practical time considerations. Transformation of meta-models expressed in UML/OCL [114] to various formal systems is not new [47] [13] [70] [99] [11] [26] [92]. In [47] the authors present a transformation from UML Class Diagrams to Description Logics. Their approach is theoretically rigourous where a knowledge base in description logic on its variants is obtained for a UML Class diagram and theorem provers such as FACT [69] and RACER [157] are used to obtain instances by inferring from the knowledge base. They prove that the time for inference using a description logic representation of an UML Class diagram in EXPTIME-complete. However, their approach does not support transformation of meta-level constraints such as those expressed in Object Constraint Language (OCL) [114] to description logic. An extension of this work for obtaining instances in finite domain is presented in [99]. The transformation of meta-level constraints such as OCL along with UML class diagrams to formal higher-order logic language called Isabelle has been explored in tools such as HOL-OCL [11]. Similarly, we have seen the transformation to constraint programming language ECLiPSE in [70]. Both, these approaches are used primarily for verification of a UML Class Diagram instance against the UMLCD metamodel specification. A constraint in OCL can be verified against an instance of UMLCD but we need the instance itself. In our pursuit to find complete models we need to automatically synthesize instances of a meta-model rather than verifying an arbitrary constraint against an existing instance.

In this thesis, we use CARTIER to transform an input metamodel, metamodel constraints and partial model to a constraint satisfaction problem in ALLOY [71]. We solve the ALLOY model to generate one or more recommendations to complete the partial model such that it contains all elements of the partial model and conforms to the metamodel and its constraints. The recommendations are brought back as high-level models in the model editor.

2.7 State of the Art in Product Discovery

In this thesis, we develop product discovery in a SPL for generation of test products. We present the related work below.

2.7.1 Related Work in SPL Test Generation

Our work deals with software-engineering specific dimensions of SPL testing: (1) scalability of test cases generation, (2) reduction of the resulting test cases set (both in terms of size of the test suite and redundancies) and (3) usability for the testers.

Concerning test generation for PL (1), McGregor [100] and Tevanlinna [150] propose a well-structured overview of the main challenges for testing product lines. A major one is obviously the exponential growth of possible products. The idea of using combinatorial testing for PL test selection is not new and has been initially proposed by Cohen et. al. [42, 41]. Combinatorial interaction testing (CIT) [39]. [90] led to the definition of pairwise testing, and then its generalization to t-wise testing. Cohen et. al. have applied CIT to systematically select configurations/products [42] that should be tested. They consider various algorithms in order to compute configurations that satisfy pair-wise and t-wise criteria [41]. The constraints imposed due to feature relationships in a feature model are solved by calling SAT solvers such as ZChaff [159]. However this approach is mainly theoretical and manual. Our work goes along the same lines but deals with scalability of the test generation, noting that CIT+SAT approaches do not scale directly with real-case feature diagrams, such as the AspectOPTIMA PL example.

Concerning test minimization for PL (2), to limit repeated testing efforts, a possible solution is to produce template system test cases, common to the whole product line and that can be adapted to each product. Nebut et al. [109] proposed a model-based approach to derive test objectives for the whole system. In [133], Scheidemann defined a method minimizing the set of configurations to verify the whole software product line. The author exploits the commonalities in order to minimize the verification effort required for requirements that pertain to several configurations. However, this approach does not take into account constraints between features which limits the applicability of the approach (see [41]). In the same vein, [160] propose a method to generate test plans covering user-specified portions of the huge number of possible configurations of a component-based software system.

Concerning the last point (3), we choose a model driven technique to automatically map a feature diagram into an Alloy input format. The user of the approach can thus manipulate directly feature digram and transform them directly in Alloy. A formalization for feature models in Alloy can be found in [124], but is not dedicated to testing and feature diagrams have to be written by hand. Uzuncoava et al. [152] use Alloy to generate a test suite incrementally from the specification of a product, directly modeled as alloy formulas. The interesting point in this work is that tests are reused from one product to another in a cumulative way. Our work focuses on testing the SPL as whole rather than individual products. Indeed, these techniques of SPL testing are complementary, our method focusing on automated selection of products, which can then be individually tested.

Usability is also a question of analysis algorithms and case tools to manipulate and reason about feature models [20, 102]. Benavides et al. have developed FAMA [21] a generic open-source framework supporting various kinds of analyses. Minimal test-set computation is not part of them but our EMF/Eclipse based T-wise toolset can be integrated easily to it. Furthermore, our variability metamodel is generic and has been successfully applied/reused for product line derivation [119] and variability weaving [105].

Chapter 3

Automatic Effective Model Discovery

In the context of Model-driven Engineering (MDE), we use first-class software artifacts called *models* to build complex software systems. A model is a graph of inter-connected objects constructed using a modelling language. For instance, the well-known Unified Modelling Language (UML) [116] is used to create models of various aspects of object-oriented software systems. The models include requirements specification using UML use case diagrams, software structure using UML class diagrams and behavior using UML activity and/or UML state machine diagrams. The set of all models specified by a modelling language is the *modelling domain* of the modelling language.

A modelling language can be very expressive and often allows the creation of an infinite number of models. The UML is one such example of a very large and expressive modelling language. The UML consists of 246 concepts with a number of properties. Infinite possible objects of these concepts can be inter-connected in a virtually infinite number of ways in models of the UML. This implies that the modelling domain of UML is an infinite set of models. Are all the models in a modelling domain useful or *effective* for a given set of objectives? The answer is no. Not all models one can construct in a modelling language are useful or *effective* given a set of objectives. There is a need for knowledge from heterogenous sources to ensure the creation of an award-winning or *effective model*.

Heterogenous sources of knowledge that can restrict the creation of models to effective models can come from different domain experts, expressed in different languages and possibly developed at different times. For example, a source based on common-sense knowledge about a modelling domain is a set of well-formedness rules for models. A textual constraint language such as the Object Constraint Language (OCL) [114] is often used to specify such well-formedness rules. An OCL invariant on the UML state machine models enforces that a state machine contains *at least one final state*. This invariant satisfies one of the requirements for correct termination of a state machine's execution. Other sources of knowledge may include partially specified models, test criteria for creation of a model for testing, a pre-condition of a model transformation that executes the model as its input and many others depending on the objective for creating the effective model. The restrictions imposed by heterogenous sources of knowledge on a modelling domain virtually leads to the notion of a subset of models in a modelling domain called the *effective modelling domain*. The effective modelling domain is most likely to contain effective models for a given set of objectives.

The creation of models in the effective modelling domain presents a pitfall. Manually creating effective models is very tedious or sometimes impossible as the modeller must simultaneously satisfy constraints from a number of sources. The magnitude of the problem becomes even more evident when we need to manually create thousands of models with an objective such as testing a model transformation. Can we partially or fully automate the process of *generating* or *discovering* effective models? This is the question that intrigues us.

We present a framework and methodology for *automatic effective model discovery* in a modelling domain. The framework is embodied in a model-driven tool CARTIER [138] [6]. The methodology is based on the general idea that an effective modelling domain can be transformed to a *constraint satisfaction problem* (CSP). Solving the constraints satisfaction problem gives us models in the effective modelling domain. However, this general idea entails a number of challenges. The three most important challenges are:

Challenge 1: Representing the modelling domain of *very large modelling* languages such as the UML as a CSP may result in a very large CSP that cannot be solved in a reasonable amount of time.

Challenge 2: Knowledge from heterogenous sources are often specified in different languages. It is therefore a challenge to *automatically transform* them to constraints in a common CSP language where modelling domain constructs may be expressed very differently.

Challenge 3: The solutions of a CSP may not be in the form of models of the initial modelling language. There is a need to *automatically transform CSP solutions back to models* in the modelling language.

Our methodology addresses these challenges in the following principal steps:

1. We automatically prune an input modelling language to obtain its effective subset

2. We transform heterogenous sources of knowledge including the pruned modelling language to a common CSP in the formal specification language ALLOY [71]

3. We solve the ALLOY model within finite bounds and automatically transform the solutions (if they exist) back as models of the input modelling language

We describe the methodology in more detail along the chapter using the running case study of generating models for the UML modelling language.

We organize the chapter as follows. In Section 3.1 we present the overall framework and methodology. In Section 3.2, we present the software embodiment CARTIER of our framework. We present the running case study of UML in Section 3.3. The first step of effective modelling domain identification via metamodel pruning is presented in Section 3.4. We describe the transformation of a basic metamodel with single inheritance to ALLOY in Section 3.5. A more complicated transformation metamodels with multiple inheritance to ALLOY is described in Section 3.6. In Section 3.7, present how we handle transformation of metamodel invariants to ALLOY. We discuss automatic model generation by solving the final ALLOY model in Section 3.9. We summarize the contents of the chapter in Section 3.12.

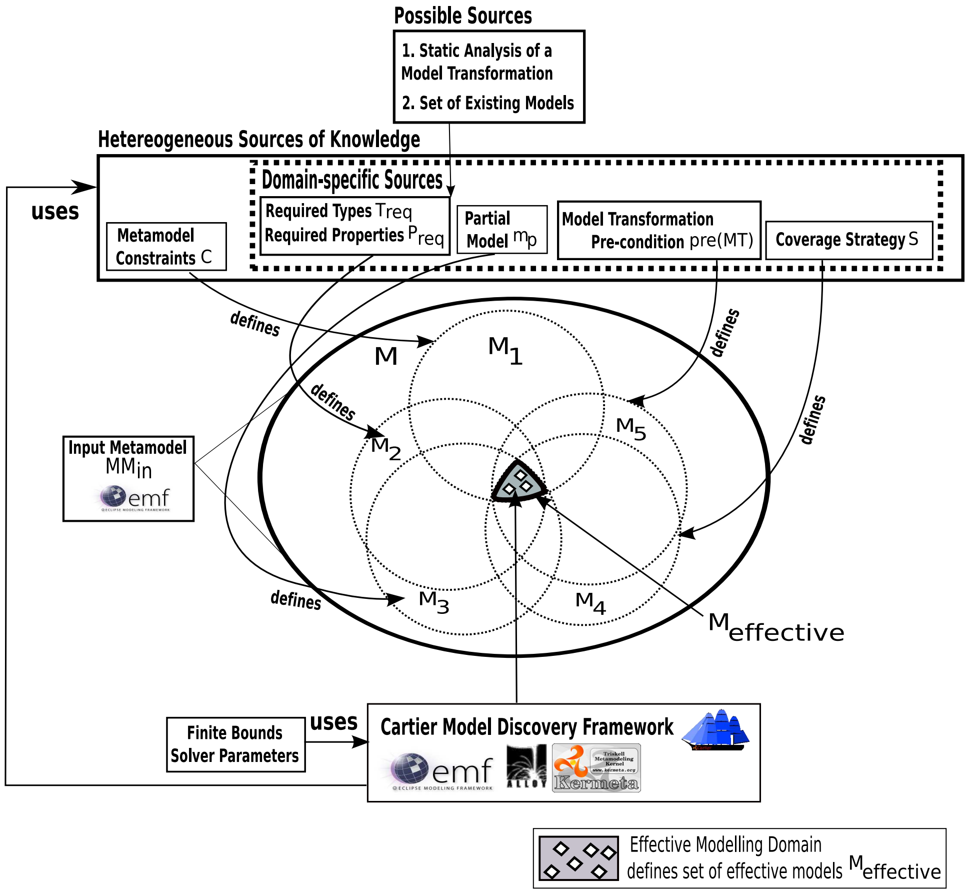

Possible Sources
1. Static Analysis of a Model Transformation
2. Set of Existing Models
Hetereogeneous Sources of Knowledge
uses
Domain-specific Sources
Metamodel Constraints C
Required Types Treq
Required Properties Preq
Partial Model mp
Model Transformation Pre-condition pre(MT)
Coverage Strategy S
defines
defines
defines
defines
defines
M
M1
M2
M3
M4
M5
Meffective
Input Metamodel MMin
emf
Finite Bounds Solver Parameters
uses
Cartier Model Discovery Framework
emf
Alloy
Kermeta
Effective Modelling Domain defines set of effective models Meffective

Figure 3.1: A Framework for Automatic Effective Model Discovery

3.1 Automatic Effective Model Discovery Framework

The framework for automatic effective model discovery is shown in Figure 3.1. The inputs to the framework include knowledge from heterogeneous sources to help specify the *effective modelling domain*. We can divide the sources of knowledge to *primary sources* and *domain-specific sources*. The general methodology followed in the framework is presented in Section 3.1.3.

3.1.1 Primary Sources of Knowledge

The primary sources of knowledge are:

- **Input metamodel** is the specification of an input modelling language. It specifies a modelling domain which is the set of *all models* in a modelling language. The input metamodel consists of a set of types (class with properties, enumeration, primitive) to instantiate objects. The industry standard framework for specifying an input metamodel is the Eclipse Modeling Framework (EMF) [58]. The input metamodel itself is stored as an instance of the ECORE metamodel.

- **Metamodel Invariants/Constraints** are expressed on an input metamodel using a textual constraint language such as Object Constraint Language (OCL) [114]. These constraints encode restrictions that cannot be specified using a diagrammatic ECORE model.

3.1.2 Domain-specific Sources of Knowledge

A number of domain-specific sources of knowledge may also help define the effective modelling domain. We present some of them below:

- **Required types and properties** in the input metamodel. The set of required types and properties help extract a subset of the input metamodel for effective model discovery. There can be many possible sources for the set of required types and properties:
  - Static analysis of a model transformation gives a set of types and properties in the input metamodel actually manipulated by the transformation.
  - A set of models conforming to the input metamodel is another source of required types and properties. Visiting the models in the set gives us a set of types and properties used in the metamodel. A typical real-world example of this could be in a classroom setting for object-oriented design using UML. The professor can point out to students the required types and properties he used to create UML by visiting every object of a set of models automatically.
- **Partial Model** is a partially specified model using the input metamodel. For instance, a graphical model editor allows an user to create models in a modelling language such as UML state machines. An incomplete model in the editor is a partial model in the UML state machine language. The partial model may not respect all metamodel constraints of UML. Therefore, a partial model is often expressed as an instance of a *relaxed version of the input metamodel*.
- **Test Coverage Strategies** help define and generate *model fragments* [55] that cover a wide range of structural aspects in the input metamodel. For instance, an input domain partition based strategy helps generate model fragments that cover partitions on all types and properties of the input metamodel. These model fragments help define an effective modelling domain for *coverage-based testing* of a model transformation. All test models that satisfy a coverage strategy contain the model fragments generated from the strategy. Model fragments are expressed in a modelling language that permits specification of ranges on properties of an input metamodel.
- **Transformation Pre-condition** is a set of invariants on the metamodel that is specific to a model transformation. A model transformation often may not be designed to transform all

models specified by its input metamodel. For instance, the transformation from class diagram models to entity relationship diagram models [22] require that all classes in the input class diagram have at least one primary attribute. The OCL [114] is often used to express pre-conditions. Generating test models requires the test models to satisfy transformation pre-conditions.

3.1.3 Methodology

The methodology for automatic effective model discovery uses the inputs presented above and can be divided in three principal steps:

1. **Effective Metamodel Identification**: We identify the effective metamodel from the input metamodel via a technique known as *metamodel pruning* [141]. Briefly, the metamodel pruning algorithm extracts a subset of the input metamodel known as the effective metamodel. The effective metamodel contains the set of required types and properties provided as input and all its obligatory dependencies. We present metamodel pruning in Section 3.4.

2. **Transformation of Effective Modelling Domain to ALLOY**: Knowledge from heterogeneous sources including the effective metamodel are transformed to a constraints model in the formal specification language ALLOY [71]. We briefly describe ALLOY in Chapter 2, Section 2.5. We describe the transformation in Sections 3.5, 3.5, 3.6, 3.7.

3. **Model Generation by solving the ALLOY Model**: We solve the ALLOY model to obtain solutions that satisfy the constraints in the ALLOY model. The solutions are transformed to models that conform to the input metamodel. In Section 3.9, we describe the process of model generation from the ALLOY model.

3.2 Software Embodiment: Cartier

We implement our framework for automatic model discovery (shown in Figure 3.1 and described in Section 3.1) in model-driven tool CARTIER. The tool was first presented in [138] and a prototype is available at [6]. It is named after *Jacques Cartier* , a french discovery and explorer from St. Malo, credited with the earliest exploration of Canadian in-lands. The construction of CARTIER has been motivated by a number of requirements as enlisted in Section 3.2.1. We describe technical aspects of CARTIER that address its requirements in Section 3.2.2.

3.2.1 Requirements for CARTIER

This section presents a number of high-level considerations that emerge while considering the implementation of a tool for *automatic model discovery* such as CARTIER.

Conformance to Industry Standards for Modelling

The widely-accepted industry standard for modelling and modeling language design is the Eclipse Modeling Framework (EMF) [32] initially developed by IBM. The metamodel of a modelling language is often available as an Ecore model of the EMF. One of the primary objectives for CARTIER is to discover or generate models that conform to a metamodel available in the Ecore format. CARTIER must be able to manipulate and transform all or most relevant aspects of the Ecore metamodels.

Sophisticated Model Manipulation and Transformation

The framework for automatic model discovery, proposed in Section 3.1, requires the implementation of a wide range of model manipulation algorithms for pruning and transformation to ALLOY. CARTIER must solicit the use of a model transformation language that supports the following important operations on models (and metamodels):

1. Scalability in loading, transforming, and saving very large metamodels and models

2. Navigation of models and creating/removing model elements

3. Support model typing to check type conformance between metamodels. We use model typing to check type conformance between the original and a pruned effective metamodel.

4. Support for invariants to express metamodel invariants and model transformation preconditions

5. Inter-operability with the high-level programming language Java. This will facilitate execution of ALLOY models

6. A model transformation language that can simultaneously manipulate models in heterogenous modelling languages

Metamodel for ALLOY

The tool must transform the effective modelling domain to a constraints model such as an ALLOY model. The transformation can be classified as a *many-to-one exogenous transformation* between models in modelling languages for heterogeneous sources to a model in ALLOY. Therefore, there is a need to create an output metamodel representing the ALLOY grammar.

3.2.2 CARTIER Technical Overview

CARTIER thrives within the context of MDE is built upon the Eclipse Modeling Framework (EMF) [32]. CARTIER is developed in Kermeta [82] [108] an executable (meta-)modelling and model transformation language developed by the TRISKELL group in INRIA, Rennes, France.

The first step in CARTIER is to obtain an effective modelling domain or a smaller effective metamodel from an input metamodel via metamodel pruning [141]. The metamodel pruning algorithm solicits large metamodel loading/saving and sophisticated model transformation operators provided by Kermeta. The effective metamodel is a subset of the input metamodel from a set-theoretic point of view and supertype of the input metamodel from a type-theoretic point of view. We use model typing [145] (see Chapter 2, Section 2.4.2) to ensure this type conformance between the effective metamodel and input metamodel. The type conformance implies that *all instances* of the effective model are instances of the input metamodel therefore preserving backward compatibility. Further, *all operations and transformations* on the effective metamodel are compatible with the possibly large input metamodel such as UML. We take note of the great advantage of pruning while dealing with large metamodels such as the UML which cannot be readily handled by a constraint solver such as ALLOY (see next paragraph). The advantage being the capability of model typing to ensure compatibility with an industry standard. Model typing is only supported in Kermeta at the time of writing making it the prime choice for the pruning transformation.

The core of CARTIER is a transformation from heterogenous sources of knowledge including the effective metamodel to the formal specification language ALLOY. This amounts to a *many-to-one exogenous* model transformation. Kermeta supports the construction of such model transformations. The heterogeneous sources of knowledge are models expressed as instances of different Ecore metamodels that can be efficiently handled by Kermeta. The target language is ALLOY [71] which is implemented in Java. To bring everything within the context of model transformation between modelling languages we created a metamodel for ALLOY conforming to the Ecore standard. The ALLOY Ecore metamodel is available for download at [2]. CARTIER navigates and extracts knowledge from the sources to create a declarative model in the language ALLOY using a Kermeta model transformation.

CARTIER must solve the ALLOY model to obtain solutions that can serve as a source of information to create model instances of the input metamodel. This calls for inter-operability with Java as the ALLOY API is in Java. Kermeta allows calling the Java API to solve the ALLOY model using relevant parameters and a SAT solver of choice such as MiniSAT [112], ZChaff [159]. The ALLOY solutions must be transformed back to model instances of the input metamodel. We present a transformation ALLOY2EMF in Java that transforms the ALLOY solutions back to model instances of ALLOY.

3.2.3 CARTIER Architecture

In Figure 3.2, we present the overall architecture of CARTIER. The architecture implements a number of model transformations as indicated by several numeric prefixes. We enlist the important steps in the architecture below:

1. Metamodel pruning (indicated as transformation 1 in Figure 3.2) transforms an input metamodel MM_{in} to the effective metamodel eMM_{in} containing the required types and properties and their obligatory dependencies. The pruning algorithm is described in Section 3.4.

2. If the effective metamodel eMM_{in} contains multiple inheritance we apply the transformation (indicated as transformation 3 in Figure 3.2) that flattens the effective metamodel to a base ALLOY model A with single inheritance. This transformation is described in Section 3.6. If the effective metamodel contains only single inheritance CARTIER executes the basic transformation (indicated as transformation 2 in Figure 3.2) to obtain the base ALLOY model A. This transformation is described in Section 3.5.

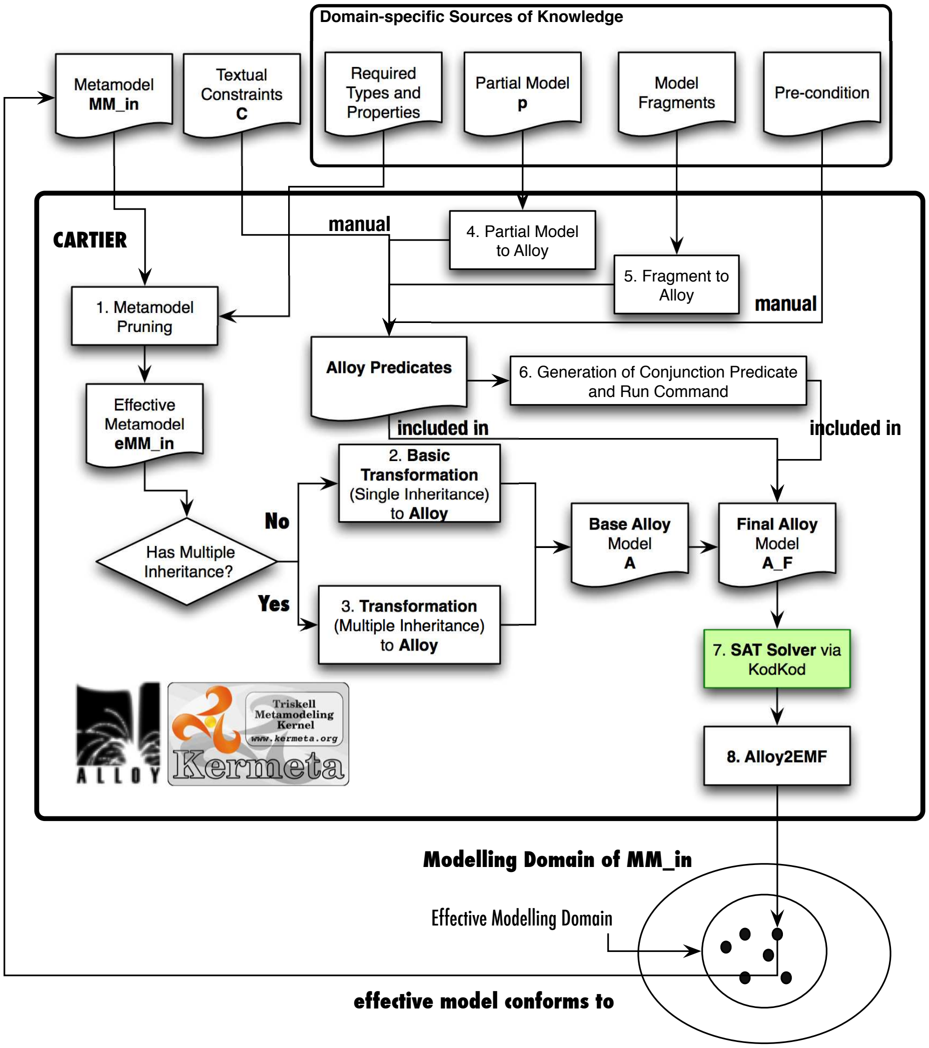

Domain-specific Sources of Knowledge
Metamodel MM_in
Textual Constraints C
Required Types and Properties
Partial Model p
Model Fragments
Pre-condition
CARTIER
manual
4. Partial Model to Alloy
5. Fragment to Alloy
manual
1. Metamodel Pruning
Alloy Predicates
6. Generation of Conjunction Predicate and Run Command
Effective Metamodel eMM_in
included in
included in
2. Basic Transformation (Single Inheritance) to Alloy
No
Has Multiple Inheritance?
Yes
Base Alloy Model A
Final Alloy Model A_F
3. Transformation (Multiple Inheritance) to Alloy
7. SAT Solver via KodKod
ALLOY
Triskell Metamodeling Kernel www.kermeta.org
Kermeta
8. Alloy2EMF
Modelling Domain of MM_in
Effective Modelling Domain
effective model conforms to

Figure 3.2: CARTIER Architecture

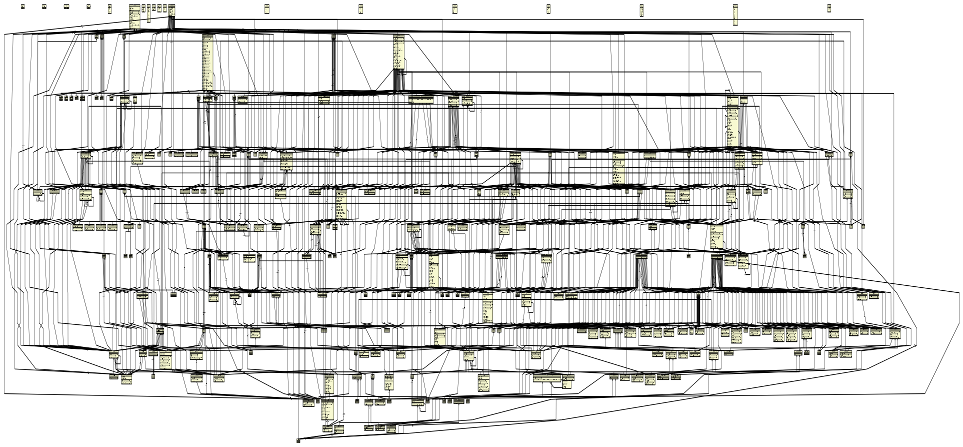


Figure 3.3: Bird’s Eye View of the UML Metamodel

3. Domain-specific sources such as the partial model p, model fragments are *automatically* transformed to ALLOY predicates using transformations 4, 5 in Figure 3.2.

4. Arbitrary textual constraints C or domain-specific knowledge such as the pre-condition of a model transformation are currently *manually* transformed to ALLOY predicates.

5. CARTIER generates a conjunction ALLOY predicate of the set of all ALLOY predicates and a corresponding run command to solve the predicate. The conjunction predicate is combined with the base ALLOY model A to give a final ALLOY model A_F. This is performed in transformation 6 as shown in Figure 3.2. The details of this transformation is presented in Section 3.9.

6. CARTIER invokes KodKod from the ALLOY API to transform the final ALLOY model A_F to a Boolean satisfaction (SAT) problem as shown in transformation 7 from Figure 3.2. This transformation already exists in the ALLOY API and is not implemented in CARTIER. CARTIER invokes a SAT solver such as ZChaff [159], or MiniSAT [112] to generate ALLOY instances.

7. The ALLOY instances are transformed to EMF models conforming to the input metamodel MM_{in}. This is depicted in transformation 8 of Figure 3.2. This transformation described in Section 3.9.3.

3.3 Running Case Study : The UML

We use the UML as a running case study to describe automatic model discovery. We present a bird’s eye view of the UML metamodel in Figure 3.3.

We believe that UML is a convincing case study to illustrate our approach for model discovery. There are a number of reasons to choose UML as a running case study:

- **Industry Standard Metamodel:** The UML is a widely accepted industry standard for software structure and behavior design and code generation. A number of model transformations have been expressed using UML as the input domain. Automatic model discovery of models in the domain of UML clearly demonstrates the applicability of our approach to real-world problems.

- **Very Large and Complex Metamodel:** The UML metamodel consists of 246 classes and 583 properties and incorporates complex metamodel patterns such as multiple containers for a class, multiple inheritance between classes, and extensive use of opposite properties in metamodels.

- **Provokes use of Sophisticated Model Transformation:** The complex structure of UML solicits the use of sophisticated model transformation operators in transformations between UML to other languages such as the Relational Database Management Systems (RDBMS) [22]. Automatic generation of test models that discover bugs in such transformations is of key interest to us.

- **Illustrates the benefits of Metamodel Pruning and Model Typing:** The large size of UML helps us demonstrates the benefits of metamodel pruning to extract a subset of UML and demonstrating type conformance of the pruned metamodel with the UML. The type conformance demonstrates that instance and operations on the subset of UML preserve backward-compatibility with UML itself.

- **Can UML be saved?** A political question that we wish to address with this case study is the growing debate about the large size of UML. Critics state that UML is evolving to become very and large incomprehensible for software development. However, they also mention that the notion of general purpose modelling languages such as UML is necessary to maintain backward compatibility and inter-operability for users. We want to demonstrate with our approach that metamodel pruning and model typing help work around the problem of the large size by extracting only relevant subsets of the UML for applications such as model discovery. All the while staying compatible with the UML standard.

3.4 Effective Modeling Domain Identification: Metamodel Pruning

We present a *metamodel pruning algorithm* that takes as input a large metamodel and a set of required classes and properties, to generate a target *effective metamodel*. The effective metamodel contains the required set of classes and properties. The term *pruning* refers to removal of unnecessary classes and properties. From a graph-theoretic point of view, given a large input graph (large input metamodel) the algorithm removes or prunes unnecessary nodes (classes and properties) to produce a smaller graph (effective metamodel). The algorithm determines if a class or property is unnecessary based on a set of rules and options. One such rule is removal of properties with lower bound multiplicity 0 and who’s type is not a required type. We demonstrate using the notion of model typing that the generated effective metamodel, a subset of the large metamodel from a set-theoretic point of view, is a *super-type*, from a type-theoretic point of view, of the large input metamodel. This means that all programs written using the effective metamodel can also be executed for models of the original large metamodel. The pruning process preserves the meta-class names and meta-property names from the large input metamodel in the effective metamodel. This also implies that all instances (models) of the effective metamodel are also instances of the initial large input metamodel. All models of the effective metamodel are exchangeable across tools that use the large input metamodel as a standard. The extracted effective metamodel is very much like a transient DSML with necessary concepts for a problem domain at a given time.

3.4.1 Important Definitions

We present some general definitions we use to describe the pruning algorithm.

Definition 3: A *metamodel MM* is a 3-tuple $MM := (T, P, Inv)$, where T is a finite set of class, primitive, and enumeration types, P is a set of properties, Inv is a finite set of invariants. We specify the modelling domain of a *modelling language* using a metamodel. We use the Ecore standard to represent a metamodel [32].

Definition 4: A *primitive type* b is an element in the set of primitives: $b \in \{String, Integer, Boolean\}$.

Definition 5: An *enumeration type* e is a 2-tuple $e := (name, L)$, where $name$ is a $String$ identifier, L is a finite set of enumerated literals.

Definition 6: A *class type* c is a 4-tuple $c := (name, P_c, Super, isAbstract, containers)$, where $name$ is a $String$ identifier, P_c is a finite set of properties of class c, class c inherits properties of classes in the finite of classes $Super$, $isAbstract$ is a $Boolean$ that determines if c is abstract and $containers$ is the set of all possible containing classes for the instances of c.

Type Operations: The operations on types used in this algorithm are: (a) $t.isInstanceOf(X)$ that returns true if t is of type X or inherits from X. (b) $t.allSuperClasses()$, if $t.isInstanceOf(Class)$, returns the set of all its super classes $t.Super$ including the super classes of its super classes and so on (multi-level) (c) $t.allContainers()$ returns all possible containers for a class type.

Definition 7: A *property* p is a 7-tuple $p := (name, oC, type, lower, upper, opposite, isComposite)$, where $name$ is a $String$ identifier, oC is a reference to the owning class type, $type$ is a reference to the property type, $lower$ is a positive integer for the lower bound of the multiplicity, $upper$ is the a positive integer for the upper bound of the multiplicity, $opposite$ is a reference to an opposite property if any, and $isComposite$ determines if the objects referenced by p are composite (No other properties can contain these objects).

Property Operations: The operation on properties in this algorithm is $p.isConstrained()$ which returns $true$ if constrained by any invariant i such that $p \in i.P_I$. This is checked for all invariants $i \in MM.Inv$.

Definition 8: An *invariant* I is a 3-tuple $c := (T_I, P_I, Expression)$, where T_I is the set of types used in the invariant I and P_I is the set of properties used in I. An $Expression$ is a function of T_I and P_I that has a boolean value. The $Expression$ is often specified in a constraint language such as OCL [114].

Note: Throughout the section, we use the *relational dot-operator* to identify an element of a tuple. For example, we want to refer to the set of all types in a metamodel we use the expression $MM.T$,or $MM.P$ to refer to the set of all properties. Also, we do not consider user-defined metamodel *operations* or its argument signatures in our approach.

3.4.2 Metamodel Pruning Algorithm

This section describes the *metamodel pruning algorithm* to transform an input metamodel to a pruned target metamodel. We acknowledge the fact there can be an entire family of pruning algorithms that can be used to prune a large metamodel to give various effective metamodels. We present a *conservative* metamodel pruning algorithm to generate effective metamodels. Our initial motivation to develop the algorithm was to help scale a formal method for test model generation [138] in the case of large input metamodels. Therefore, given a set of required classes and properties the rationale for designing the algorithm was to remove a maximum number of classes and properties facilitating us to scale a formal method to solve constraints from a relatively small input metamodel. The set of required classes and properties are inputs that can come from either static analysis of a transformation, an example model, an objective function, or can be manually specified. Given these initial inputs we automatically identify mandatory dependent classes and properties in the metamodel and remove the rest. For instance, we remove all properties which have a multiplicity 0..* and with a type not in the set of required class types. However, we also add some flexibility to the pruning algorithm. We provide options such as those that preserve properties (and their class type) in a required class even if they have a multiplicity 0..*. In our opinion, no matter how you choose to design a pruning algorithm the final output effective metamodel should be a supertype of the large input metamodel. The pruning algorithm must also preserve identical meta-concept names such that all instances of the effective metamodel are instances of the large input metamodel. These final requirements ensure backward compatibility of the effective metamodel with respect to the large input metamodel.

Algorithm Overview

In Figure 3.4, we present an overview of the metamodel pruning algorithm. The inputs to the algorithm are: (1) A source metamodel $MM_s = MM_{large}$ which is also a large metamodel such as the metamodel for UML with about 246 Classes and 583 properties (in Ecore format) (2) A set of required classes C_{req} (3) A set of required properties P_{req}, and (4) A boolean array consisting of parameters to make the algorithm flexible for different pruning options.

The set of required classes C_{req} and properties P_{req} can be obtained from various sources as shown in Figure 3.4: (a) A static analysis of a model transformation can reveal which classes and properties are used by a transformation (b) The sets can be directly specified by the user (c) A test objective such as a set of partitions of the metamodel [55] is a specified on different properties which can be source for the set P_{req}. (d) A model itself uses objects of different classes. These classes and their properties can be the sources for C_{req} and P_{req}.

The output of the algorithm is a pruned effective metamodel $MM_t = MM_{effective}$ that contains all classes in C_{req}, all properties in P_{req} and their associated dependencies. Some of the dependencies are mandatory such as all super classes of a class and some are optional such as

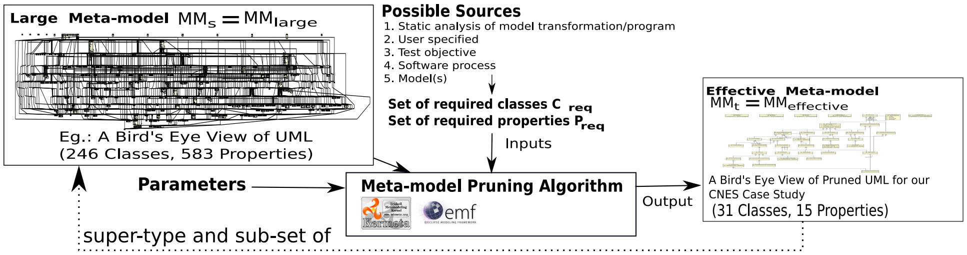

Large Meta-model MMs = MMlarge
Eg.: A Bird's Eye View of UML
(246 Classes, 583 Properties)
Possible Sources
1. Static analysis of model transformation/program
2. User specified
3. Test objective
4. Software process
5. Model(s)
Set of required classes C req
Set of required properties Preq
Inputs
Parameters
Meta-model Pruning Algorithm
emf
Output
Effective Meta-model
MMt = MMeffective
A Bird's Eye View of Pruned UML for our CNES Case Study
(31 Classes, 15 Properties)
super-type and sub-set of

Figure 3.4: The Meta-model Pruning Algorithm Overview

properties with multiplicity 0..* and whose class type is not in C_{req}. A set of parameters allow us to control the inclusion of these optional properties or classes in order to give various effective metamodels for different applications. The output metamodel $MM_{effective}$ is a subset and a super-type of MM_s.

The Algorithm

The metamodel pruning algorithm (shown in Algorithm 1) has four inputs: (a) A source metamodel MM_s (b) Initial set of required types T_{req} (c) Initial set of required properties P_{req} (d) The top-level container class type C_{top}. (e) $Parameter$ which is a Boolean array. Each element in the array corresponds to an option to add classes or properties to the required set of classes and properties. We consider three such options giving us a $Parameter$ vector of size 3.

The output of the algorithm is the pruned target metamodel MM_t. We briefly go through the working of the algorithm. The target metamodel MM_t is initialized with the source metamodel MM_s. The algorithm is divided into three main phases: (1) Computing set of all required types T_{req} in the metamodel ,(2) Set of all required properties P_{req} in the metamodel (3) Removing all types and properties not that are not in T_{req} and P_{req}

The first phase of the algorithm involves the computation of the entire set of required types T_{req}. The initial set T_{req} is passed as a parameter to the algorithm. We add the top-level container class C_{top} of MM_s to the set of required types T_{req} as shown in Step 2. In Step 3, we add the types of all required properties P_{req} to the set of required types T_{req}. In Step 4, we add types of all mandatory properties to T_{req}. Types of all properties with *lower bound greater than zero* are added to the set of required types T_{req} (Step 4.1). Similarly, if a property is constrained by an invariant in $MM.Inv$ then its type is included in T_{req} as shown in Step 4.2. If a property has an opposite type then we include the type of the opposite property to T_{req} in Step 4.3. The algorithm provides three options to add types of properties with lower multiplicity zero and are of type Class, PrimitiveType, and Enumeration respectively. The inclusion of these types is depicted in Steps 4.4, 4.5, and 4.6. The truth values elements of the $Parameter$ array determine if these options are used. These options are only examples of making the algorithm flexible. The $Parameter$ array and the options can be extended with general and user-specific requirements for generating effective metamodels. After obtaining T_{req} we add all its super classes across all levels to the set T_{req} as shown in Step 5.

The second phase of the algorithm consists of computing the set of all required properties

Required Properties	Required Types
Property : Type	Class
memberEnd : Class	Association
general : Class	Package
ownedEnd : Class	Property
classifier : Package	PrimitiveType
datatype : Property	
attribute : Class	
packagedElement : Package	

Table 3.1: Required UML Types and Properties in the Transformation class2rdbms

P_{req}. Inclusion of mandatory properties are depicted from Step 6.1 through Step 6.5. In Step 6.1, we add all properties whose type are in T_{req} to P_{req}. In Step 6.2 we add all properties whose owning class are in T_{req} to P_{req}. In Step 6.3, we add properties with lower multiplicity greater than zero to P_{req}. If a property is constrained by a constraint in $MM.Inv$ we add it to P_{req} as depicted in Step 6.4. We add the opposite property of a required property to P_{req}. Finally, based on the options specified in the $Parameter$ array, the algorithm adds properties to P_{req} with lower multiplicity zero and other characteristics.

In the third phase of the algorithm we remove types and properties from MM_t. In Step 7, we remove all properties that are not in P_{req} (Step 7.1) and all properties who's types are not in T_{req} (Step 7.2). In Step 8, we remove all types not in T_{req}. The result is an effective metamodel in MM_t. In Chapter 2, Section 2.4.2, we present *model typing* for metamodels to show that MM_t is a super-type of MM_s. As a result, any program written with MM_t can be executed using models of MM_s.

3.4.3 Illustration on UML Case Study

We prune the UML metamodel based on a set of required types and properties using the pruning algorithm. The source for the set of required types and properties is the static analysis of a model transformation between UML class diagrams and Relational Database Management Systems models described in [22]. We enlist the set of required types and properties in Table 3.1.

The pruned UML metamodel contains 26 Classes and 65 Properties which is drastically smaller than the original 246 Classes and 583 Properties. We also verify using model typing (see Chapter 2, Section 2.4.2) that the pruned UML is a supertype of UML. This implies that any model created as an instance of the pruned UML is also an instance of the original UML. Any operation or model transformation written for UML is also applicable to UML. The pruned metamodel is an effective metamodel of UML that will be used as an example for the subsequent sections.

3.4.4 Validity and Complexity of the Algorithm

The metamodel pruning algorithm by construction generates an effective metamodel that is a *supertype* of the large input metamodel. Does the algorithm generate a supertype effective metamodel for any input metamodel and set of required types and properties? We need to answer

Algorithm 1 metamodelPruning(MM_s, T_{req}, P_{req}, C_{top}, $Parameter$)

```
1. Initialize target meta-model $MM_t$
$MM_t \leftarrow MM_s$
2. Add top-level class into the set of required types
$T_{req} \leftarrow T_{req} \cup C_{top}$
3. Add types of required properties to set of required types
$P_{req}.each\{p|T_{req} \leftarrow T_{req} \cup p.type\}$
4. Add types of obligatory properties
$MM_t.P.each\{p|$
4.1 $(p.lower > 0) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
4.2 $(p.isConstrained(MM_t.Inv)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
4.3 $(p.opposite! = Void) \implies \{T_{req} \leftarrow T_{req} \cup p.opposite.type\}$
Option 1: Property of type Class with lower bound 0
if $Parameter[0] == True$ then
    4.4 $(p.lower == 0\, and\ p.type.isInstanceOf(Class)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
end if
Option 2: Property of type PrimitiveType with lower bound 0
if $Parameter[1] == True$ then
    4.5 $(p.lower == 0\, and\ p.type.isInstanceOf(PrimitiveType)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}$
end if
Option 3: Property of type Enumeration with lower bound 0
if $Parameter[2] == True$ then
    4.6 $(p.lower == 0\, and\ p.type.isInstanceOf(Enumeration)) \implies \{T_{req} \leftarrow T_{req} \cup p.type\}\}$
end if
5. Add all multi-level super classes of all classes in $T_{req}$
$MM_t.T.each\{t\,|\,t.isInstanceOf(Class) \implies t.allSuperClasses.each\{s|T_{req} \leftarrow T_{req} \cup s\}\}$
6. Add all required properties to $P_{req}$
$MM_t.P.each\{p|$
6.1 $(p.type \in T_{req}) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.2 $(p.oC \in T_{req}) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.3 $(p.lower > 0) \implies P_{req} \leftarrow P_{req} \cup p\}$
6.4 $(p.isConstrained(MM_t.Inv)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
6.5 $(p.opposite! = Void) \implies P_{req} \leftarrow P_{req} \cup p.opposite\}$
Option 1: Property of type Class with lower bound 0
if $Parameter[0] == True$ then
    6.6 $(p.lower == 0\, and\ p.type.isInstanceOf(Class)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
end if
Option 2: Property of type PrimitiveType with lower bound 0
if $Parameter[1] == True$ then
    6.7 $(p.lower == 0\, and\ p.type.isInstanceOf(PrimitiveType)) \implies \{P_{req} \leftarrow P_{req} \cup p\}$
end if
Option 3: Property of type Enumeration with lower bound 0
if $Parameter[2] == True$ then
    6.8 $(p.lower == 0\, and\ p.type.isInstanceOf(Enumeration)) \implies \{P_{req} \leftarrow P_{req} \cup p\}\}$
end if
7. Remove Properties
$MM_t.P.each\{p|$
7.1 $p \notin P_{req} \implies (t.P \leftarrow t.P - p)$
$\}$
8. Remove Types
$MM_t.T.each\{t|t \notin T_{req} \implies MM_t.T \leftarrow MM_t.T - t\}$
```

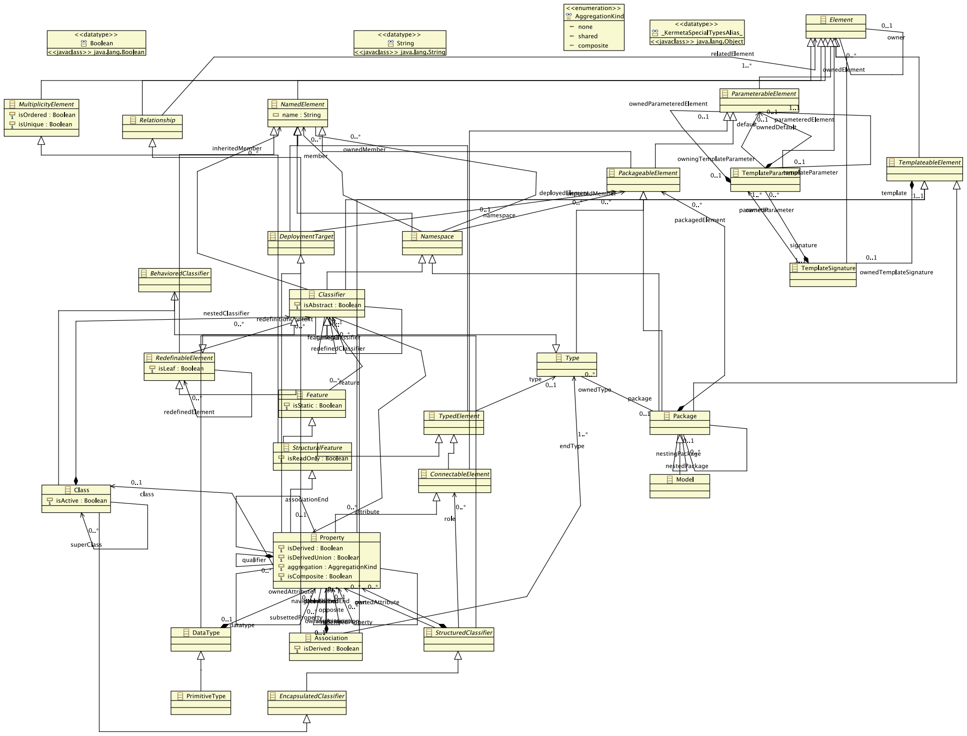


Figure 3.5: Bird’s Eye View of UML Pruned With 26 Classes and 65 Properties

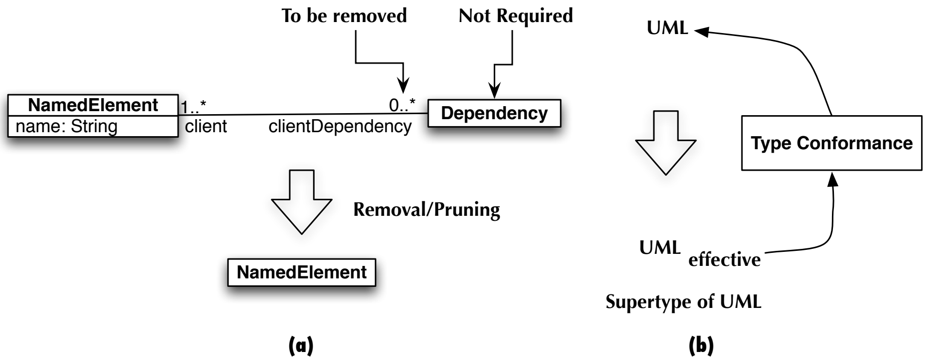


Figure 3.6: Validation of Pruning Operators (a) Operator to Remove a Property with Multiplicity 0..* (b) Model Type Conformance

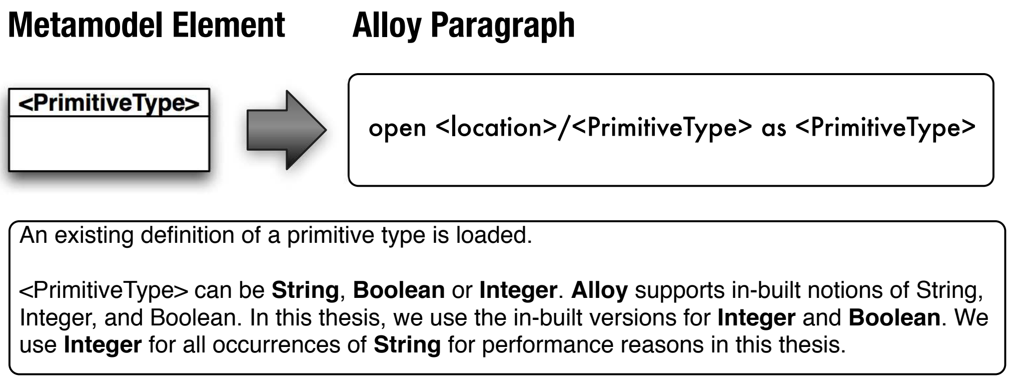
to answer this question to ensure that the algorithm is returns a super type for all possible input metamodels.

To answer this question we need to verify that each *pruning operator* takes as input a metamodel and returns a *supertype* metamodel as output. In our algorithm, each removal or pruning operator satisfies this requirement. For example in Figure 3.6 we illustrate the operator to remove a property with multiplicity 0..* of a property with a not required type. Specifically, we show that in the UML metamodel the property *clientDependency* of NamedElement may be removed when Dependency is not one of the required classes in Figure 3.6 (a). The resulting effective UML metamodel is supertype of the UML metamodel.

Similarly, we verify that all removal/pruning operators in our algorithm give a supertype as output. Therefore, by the *law of transitivity* executing the pruning operators in sequence always gives a supertype as the output.

The metamodel pruning algorithm has *linear time complexity*. The algorithm traverses the metamodel three times. The metamodel is usually a graph data structure but an Ecore metamodel enforces a containment relationship for all types. This means that a metamodel may be traversed like a tree. If a metamodel has P properties (leaf elements) then a depth-first traversal has complexity $O(P)$. The second traversal requires identification of dependent of properties and types. Finally, the third traversal removes or prunes the properties and types that are not required. Therefore, in general the time complexity of the algorithm is $O(3P)$. However, if the metamodel contains E enumerations then the complexity becomes $O(3P+E)$.

The *space complexity* of the metamodel pruning algorithm corresponds to the length *longest path* from the root of a metamodel to its root. This corresponds to the path from the root class to the property node in the last class of the containment hierarchy of a metamodel. The depth-first algorithm stores this path in memory each time it traverses a branch in the metamodel.



An existing definition of a primitive type is loaded.

<PrimitiveType> can be **String**, **Boolean** or **Integer**. **Alloy** supports in-built notions of String, Integer, and Boolean. In this thesis, we use the in-built versions for **Integer** and **Boolean**. We use **Integer** for all occurrences of **String** for performance reasons in this thesis.

Figure 3.7: Transformation of Primitive Types

3.5 Transformation Metamodel with Single Inheritance to ALLOY

In the previous section we obtain a concise and effective metamodel $MM_{effective}$ from the input metamodel MM_{in}. We now describe the transformation of the effective metamodel $MM_{effective}$ to ALLOY. For convenience, we denote the effective metamodel as just metamodel MM.

A *metamodel MM* is a 3-tuple $MM := (T, P, Inv)$, where T is a finite set of class, primitive, and enumeration types, P is a set of properties, Inv is a finite set of invariants. We use an example-driven approach to explain the transformation of each of these metamodel elements in the following paragraphs. In this section, we consider the simplest form of transformation where the metamodel contains only *single inheritance* and not multiple inheritance.

3.5.1 Transformation of a Primitive Type to ALLOY

Primitive Type Rule 1 (PTR1): We transform a primitive type such as Boolean, Integer, and String by loading in-built ALLOY modules containing specifications of Boolean and Integer. At the time of implementing the transformation we created an ALLOY model of String. The complete ALLOY string specification may be downloaded at the site [5]. However, generating strings using ALLOY is computationally expensive. Our focus is model generation with emphasis on facilitating generation of complex structural aspects of the model. Therefore, we make the choice of replacing all String properties with Integer values.

3.5.2 Transformation of an Enumeration Type to ALLOY

Enumeration Type Rule 2 (ETR2): An enumeration type such as EnumerationA in Figure 3.8 is very simply and directly transformed to an ALLOY enumeration.

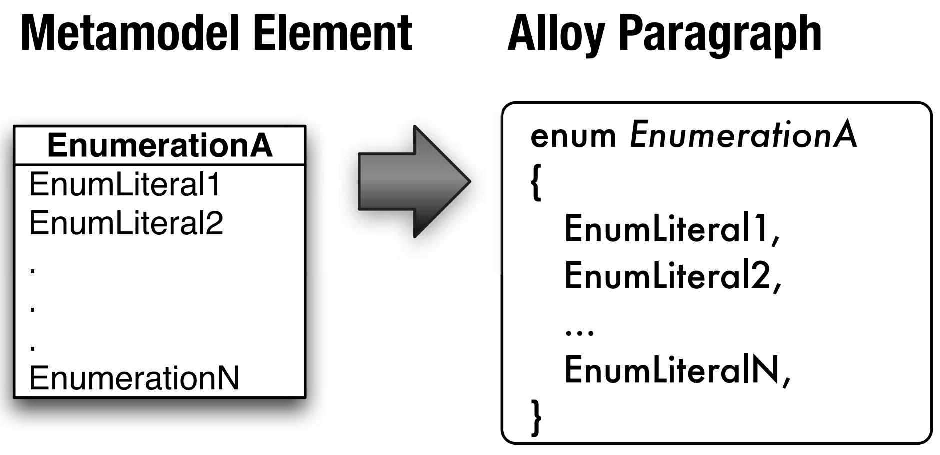


Figure 3.8: Transformation of Enumeration Type

3.5.3 Transformation of a Class Type to ALLOY

There are four specific cases in transforming a class type to ALLOY as seen in Figure 3.9. We describe them below:

Concrete Class Type With No Inheritance Rule 3 (CCNI3): A concrete class ClassA that does not inherit from any other class is transformed to an ALLOY signature. See Figure 3.9 (a).

Abstract Class Type With No Inheritance Rule 4 (ACNI4): A abstract class ClassA that does not inherit from any other class is transformed to an abstract ALLOY signature. See Figure 3.9 (b).

Concrete Class Type With Single Inheritance Rule 5 (CCSI5): A concrete class ClassA that inherits from exactly one super class SuperClass is transformed to an ALLOY signature that extends the signature representing the super class SuperClass. See Figure 3.9 (c).

Abstract Class Type With Single Inheritance Rule 6 (ACSI6): An abstract class ClassA that inherits from exactly one super class SuperClass is transformed to an ALLOY signature that extends the signature representing the super class SuperClass. See Figure 3.9 (d).

3.5.4 Transformation of a Property to ALLOY

A *property* in a metamodel is either an *attribute* pointing to primitive type a or a *reference* to object(s) of an other class. There are six specific cases to transform properties in a metamodel to *fields* in ALLOY signatures:

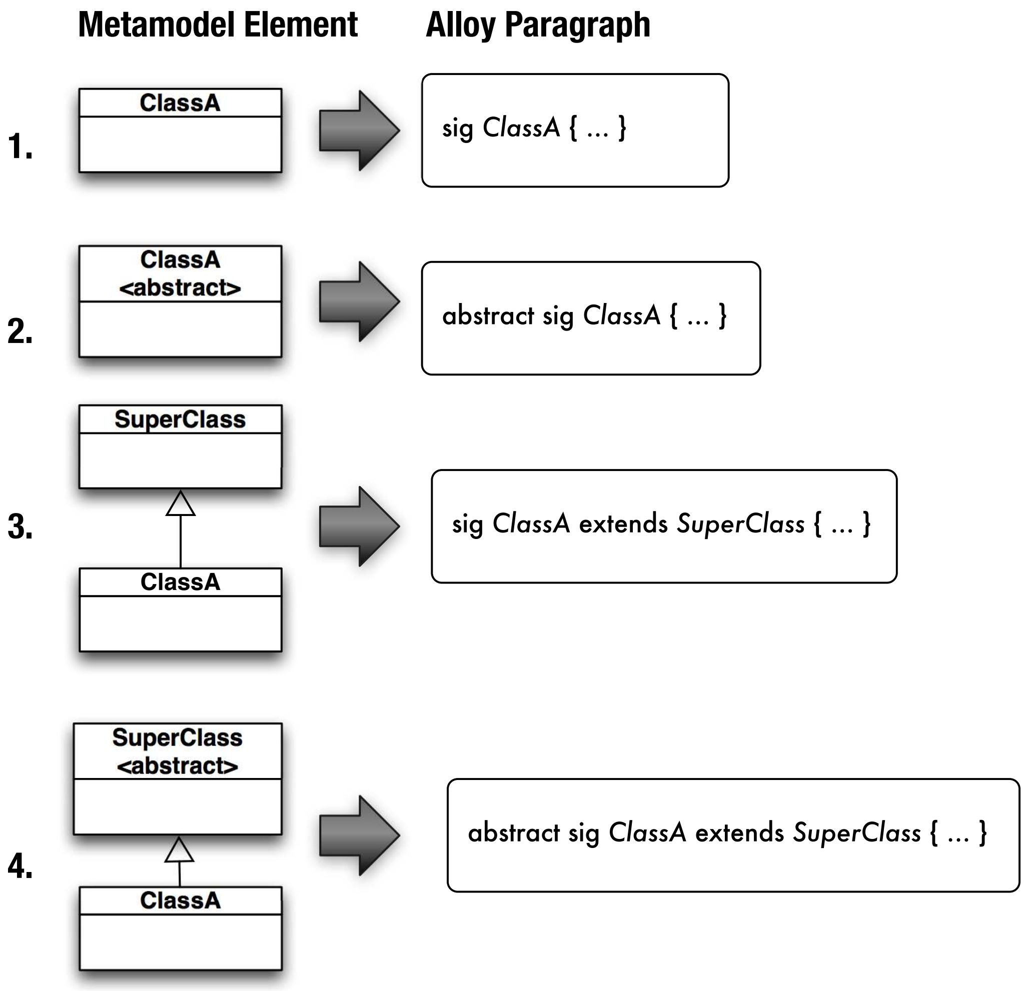


Figure 3.9: Transformation of Class Type

Primitive Attribute with One Multiplicity Rule 7 (PAOM7): Primitive attributes such as attribute1, attribute2, attribute3 with both lower and upper bound multiplicity 1 as shown in Figure 3.10 (a) are transformed to ALLOY fields with the same name. Note that the usage of primitive types had already led to the inclusion of in-built ALLOY modules implementing the definition of the primitive types. The attribute attribute3 of the String type is transformed to an ALLOY Integer in this thesis to avoid extra computational cost due to generation of strings.

Primitive Attribute with At Least One Multiplicity Rule 8 (PALOM8) : A primitive attribute attribute1 with lower bound multiplicity 0 and upper bound multiplicity 1 is transformed to an ALLOY field in its owning signature with the *lone* specialization as shown in Figure 3.10 (b).

Primitive Attribute with Variable Multiplicity Rule 9 (PAVM9) : A primitive attribute attribute1 with lower bound multiplicity a and upper bound multiplicity b, where $a \geq 0, b > a, b \neq 1$ is transformed to an ALLOY field in its owning signature with the *set* specialization as shown in Figure 3.10 (c).

Reference with One Multiplicity Rule 10 (ROM10) : A reference reference1 with lower and upper bound multiplicities 1 is transformed to an ALLOY field in its owning signature with the *one* specialization as shown in Figure 3.10 (d).

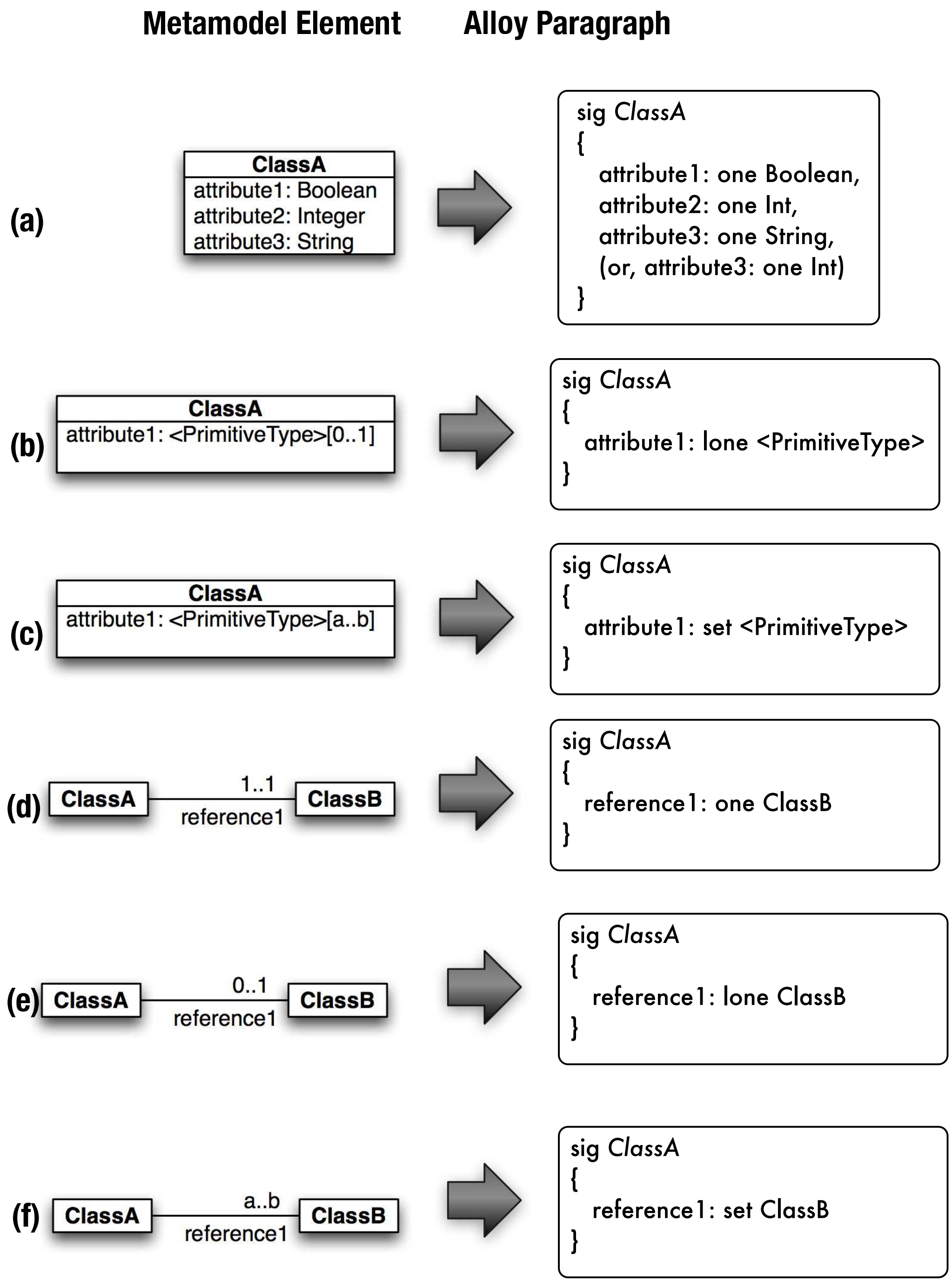
Reference with At Least One Multiplicity Rule 11 (RLOM11) : A reference reference1 with lower bound multiplicity 0 and upper bound multiplicity 1 is transformed to an ALLOY field in its owning signature with the *lone* specialization as shown in Figure 3.10 (e).

Reference with Variable Multiplicity Rule 12 (RVOM12) : A reference reference1 with lower bound multiplicity a and upper bound multiplicity b, where $a \geq 0, b > a, b \neq 1$, is transformed to an ALLOY field in its owning signature with the *set* specialization as shown in Figure 3.10 (f).

3.5.5 Transformation of Implicit Metamodel Constraints to ALLOY Facts

There are a number of constraints encoded in the input metamodel. These include constraints due to multiplicity, opposite properties, identity properties, composite properties, and containment. These implicit constraints are automatically transformed to ALLOY facts. We describe the transformation of each fact below:

Primitive Attribute Multiplicity Constraint Rule 13 (PAMC13): A primitive attribute attribute1 in a ClassA with a lower bound multiplicity a and an upper bound multiplicity b, where $a \geq 0, b > a, b \neq 1$ results in the generation of an ALLOY fact as shown in Figure 3.11 (a). The ALLOY fact states that for all objects of type ClassA the size of (denoted by #) ClassA.attribute1 must be $\geq$ attribute1.$lower$ and $\leq$ attribute1.$upper$.



```
sig ClassA
{
  attribute1: one Boolean,
  attribute2: one Int,
  attribute3: one String,
  (or, attribute3: one Int)
}
```

```
sig ClassA
{
  attribute1: lone <PrimitiveType>
}
```

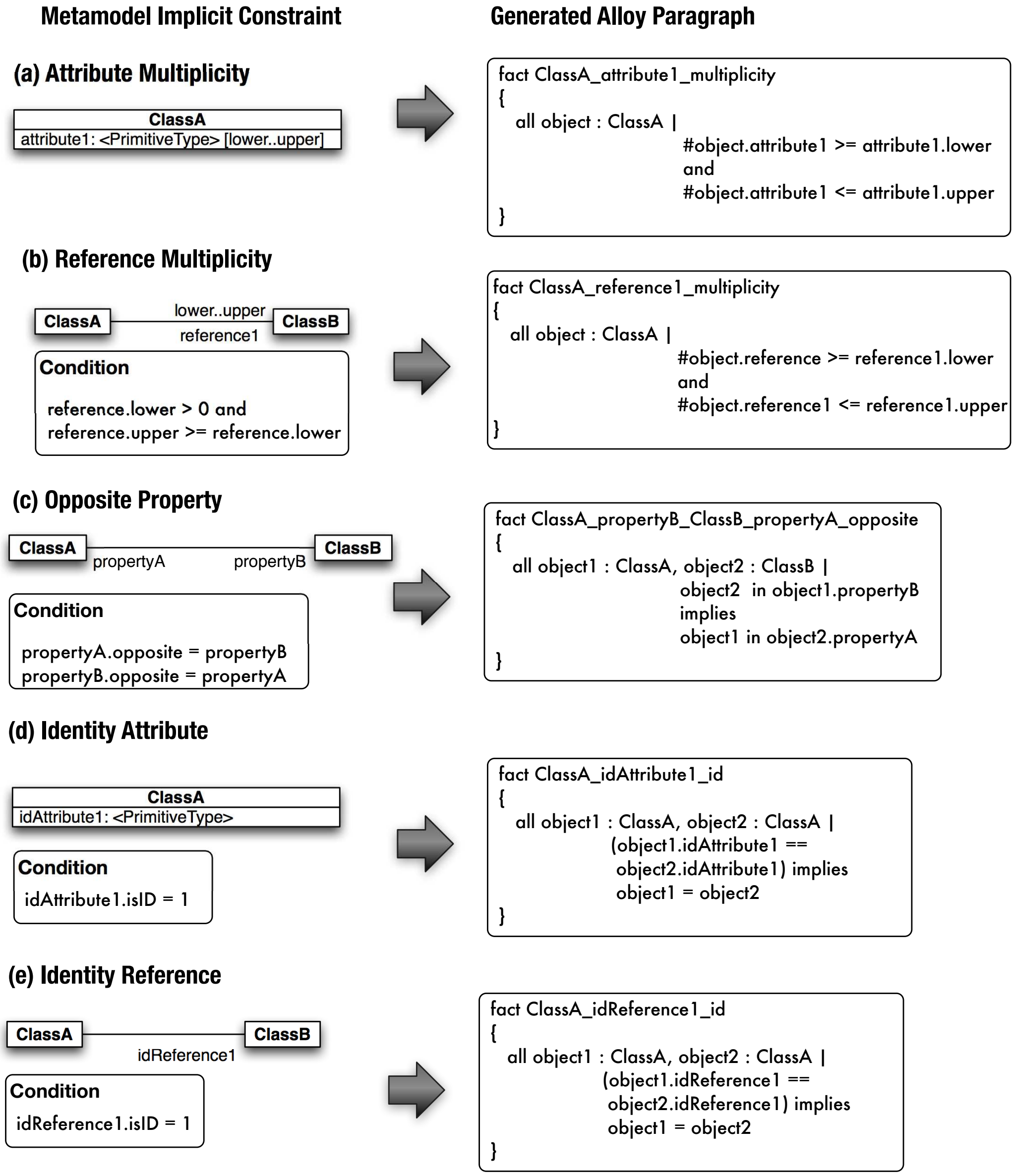
```
sig ClassA
{
  attribute1: set <PrimitiveType>
}
```

```
sig ClassA
{
  reference1: one ClassB
}
```

```
sig ClassA
{
  reference1: lone ClassB
}
```

```
sig ClassA
{
  reference1: set ClassB
}
```

Figure 3.10: Transformation of Properties to ALLOY


Metamodel Implicit Constraint
Generated Alloy Paragraph
(a) Attribute Multiplicity
ClassA
attribute1: <PrimitiveType> [lower..upper]
(b) Reference Multiplicity
ClassA
lower..upper
reference1
ClassB
Condition
reference.lower > 0 and
reference.upper >= reference.lower
(c) Opposite Property
ClassA
propertyA
propertyB
ClassB
Condition
propertyA.opposite = propertyB
propertyB.opposite = propertyA
(d) Identity Attribute
ClassA
idAttribute1: <PrimitiveType>
Condition
idAttribute1.isID = 1
(e) Identity Reference
ClassA
idReference1
ClassB
Condition
idReference1.isID = 1

```
fact ClassA_attribute1_multiplicity
{
   all object : ClassA |
                           #object.attribute1 >= attribute1.lower
                           and
                           #object.attribute1 <= attribute1.upper
}
```

```
fact ClassA_reference1_multiplicity
{
   all object : ClassA |
                           #object.reference >= reference1.lower
                           and
                           #object.reference1 <= reference1.upper
}
```

```
fact ClassA_propertyB_ClassB_propertyA_opposite
{
   all object1 : ClassA, object2 : ClassB |
                           object2  in object1.propertyB
                           implies
                           object1 in object2.propertyA
}
```

```
fact ClassA_idAttribute1_id
{
   all object1 : ClassA, object2 : ClassA |
                 (object1.idAttribute1 ==
                  object2.idAttribute1) implies
                  object1 = object2
}
```

```
fact ClassA_idReference1_id
{
   all object1 : ClassA, object2 : ClassA |
                 (object1.idReference1 ==
                  object2.idReference1) implies
                  object1 = object2
}
```

Figure 3.11: Transformation of Implicit Constraints in Metamodel to ALLOY Part 1

Reference Multiplicity Constraint Rule 14 (RMC14) : A reference reference1 in a ClassA with a lower bound multiplicity a and an upper bound multiplicity b, where $a \geq 0, b > a, b \neq 1$ results in the generation of an ALLOY fact as shown in Figure 3.11 (b). The ALLOY fact states that for all objects of type ClassA the size of (denoted by #) ClassA.reference1 must be $\geq$ reference1.$lower$ and $\leq$ reference1.$upper$.

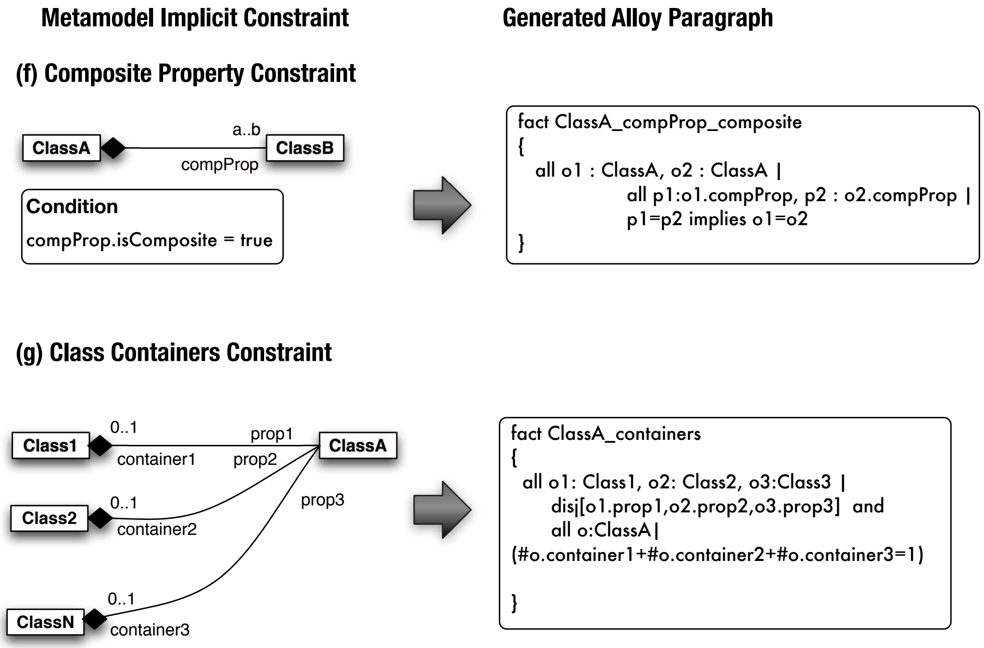
Opposite Property Constraint Rule 15 (OPC15) : Bi-directional references in a metamodel are modelled using the notion of *opposite properties*. For instance, in Figure 3.11 (c), ClassA.propertyB and ClassB.propertyA are opposite properties that lead to the generation of an ALLOY fact. The fact states that for each object object1 of ClassA and each object object2 of ClassB, if object2 is in the set ClassA.propertyB then object1 is in the set ClassB.propertyA. This fact ensures the opposite property relationship between all opposite properties in instance models of the metamodel.

Identity Attribute Constraint Rule 16 (IAC16) : An identity attribute idAttribute1 of primitive type in a ClassA as shown in Figure 3.11 (d), is transformed to an ALLOY fact. The fact states that for each object object1, object2 both of ClassA, if object1.idAttribute1 $=$ object2.idAttribute1 , then the objects object1 and object2 must be the same objects. There cannot exist two or more instances of these objects object1 and object2. The identity attribute is useful in creating objects with one or more unique identifier attributes.

Identity Reference Constraint Rule 17 (IRC17) : An identity reference isReference1 in a ClassA referring to ClassB as shown in Figure 3.11 (e), is transformed to an ALLOY fact. The fact states that for each object object1, object2 both of ClassA, if object1.isReference1 $=$ object2.isReference1 , then the objects object1 and object2 must be the same objects. There cannot exist two or more instances of these objects object1 and object2.

Composite Property Constraint Rule 18 (CPC18) : The composite property ClassA.compProp in a class ClassA containing objects of ClassB is transformed to an ALLOY fact as shown in Figure 3.12 (f). The fact states that for all objects o1, o2 of ClassA and for each reference p2 and p2 in ClassA.o1.compProp, if p2 and p2 are the same then objects o1 and o2 are the same. The fact simply states that an object of ClassB is contained in exactly one object of ClassA.

Class Containers Constraint Rule 19 (CCC19) : Objects of a class can have many possible containers. For instance, in Figure 3.12(g) the ClassA has 3 possible containers Class1, Class2, and Class3. The multiplicity 0..1 for reference ClassA.container1 indicates that Class1 may or may not be a container for ClassA objects. Similarly, the multiplicities of ClassA.container2 and ClassA.container3 indicate that Class2 and Class3 are other possible containers for ClassA objects. In a model of the metamodel a given ClassA object can be contained only in one of the three classes Class1, Class2, and Class3. This case can be extended to N possible container classes. We generate an ALLOY fact to enforce this containment relationship between objects of classes. The fact states that for all objects ob1, ob2, and ob3 of type Class1, Class2, and Class3 respectively, the reference to ClassA is *disjoint* or ob1, ob2, and ob3 always refer to different ob-



Metamodel Implicit Constraint

Generated Alloy Paragraph

(f) Composite Property Constraint

```
fact ClassA_compProp_composite
{
  all o1 : ClassA, o2 : ClassA |
              all p1:o1.compProp, p2 : o2.compProp |
              p1=p2 implies o1=o2
}
```

(g) Class Containers Constraint

```
fact ClassA_containers
{
  all o1: Class1, o2: Class2, o3:Class3 |
      disj[o1.prop1,o2.prop2,o3.prop3]  and
      all o:ClassA|
(#o.container1+#o.container2+#o.container3=1)

}
```

Figure 3.12: Transformation of Implicit Constraints in Metamodel to ALLOY Part 2

jects of ClassA. The fact also states in conjunction that all objects of ClassA must be contained in either Class1, Class2 or Class3 but never in more than one class.

3.6 Transforming Metamodel with Multiple Inheritance to ALLOY

The basic transformation, discussed in the previous Section 3.5, of a metamodel to ALLOY is suitable for small metamodels with single inheritance. However, it is not appropriate for large and complex metamodels containing several hundreds classes and properties along with complex structure such as multiple inheritance. The UML is a notorious example of a metamodel with several instances of multiple inheritance and a large number of classes (246) and properties (583). The basic transformation does not handle multiple inheritance and may result in the generation of an intractable ALLOY model when the number of classes and properties is very high.

In this section, we consider a very general case of transforming any metamodel with multiple inheritance to ALLOY for the purpose of model synthesis. Our transformation is based on the following important observations:

- Given a metamodel, a modeller creates an instance model by only creating objects of **concrete classes** in the metamodel.

- All properties have either primitive values such as integer, boolean, string or refer to objects of other concrete classes.
- Even properties referring to abstract classes ultimately point to objects of concrete classes that inherit from these abstract classes.
- A model is always an interconnected graph of *concrete class objects*.
- Abstract class objects are never created! All we need are concrete class objects and build relationships between them.

The transformation from a large and complex metamodel with multiple inheritance to ALLOY is based on the observations made above. In the subsequent sections, we present the transformation of a metamodel to a tractable and small ALLOY model. The ALLOY model uniquely contains signatures for concrete classes in the metamodel. A number of ALLOY facts are generated to emulate multiple inheritance and its effects on classes and properties in ALLOY.

3.6.1 Flattening the Class Hierarchy

Before the flattening step, all primitive types detected in the metamodel are transformed to ALLOY open statements that load modules for primitive types such as Integer, Boolean, and String. This process is exactly the same as described in Section 3.5 for transformation of primitive types.

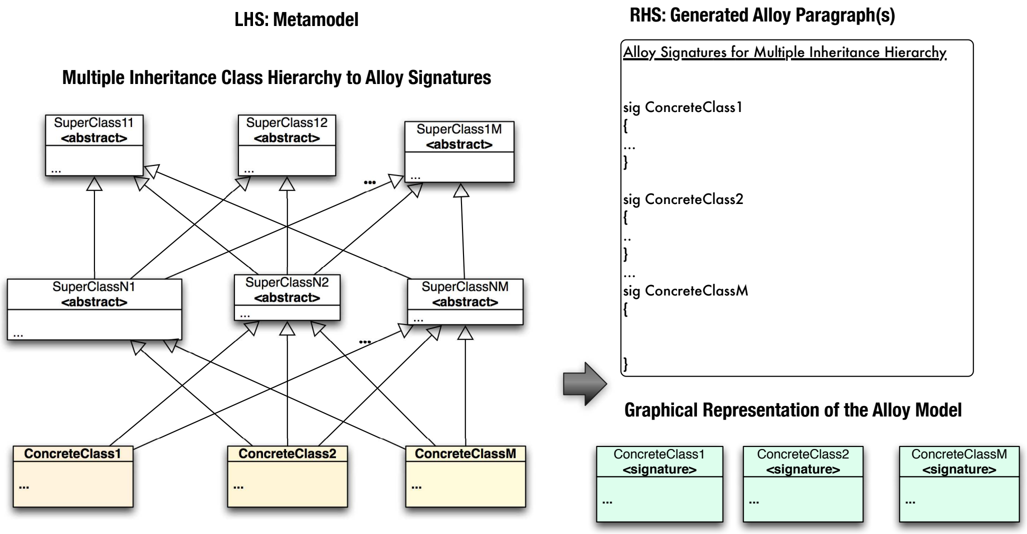
The first step in the transformation involves flattening the class hierarchy in a metamodel with multiple inheritance to a flat ALLOY model. Consider a general metamodel as shown on the left hand side of Figure 3.13 containing several abstract classes and concrete classes. As seen on the right hand side Figure 3.13, we transform all *concrete classes* in the metamodel to signatures in ALLOY. We also see the graphical signature hierarchy representation of the ALLOY model. In the figure the concrete classes ConcreteClass1, ConcreteClass2,..., ConcreteClassM are transformed to ALLOY signatures. **None** of the abstract superclasses SuperClass11...SuperClassNM are transformed to ALLOY signatures. We neglect super classes based on the observation that we will never need to instantiate objects of these abstract super classes.

3.6.2 Transforming Properties to ALLOY Fields and ALLOY Facts

The second step involves the transformation all properties of each concrete class to ALLOY. These properties include those what were originally owned by a concrete class and those inherited from all abstract classes. Therefore, we transform each property p (owned or inherited from abstract classes) in each concrete class C to ALLOY. We need to deal with the following cases:

1. **Owned Property p is of Primitive Type in a Concrete Class C:**

   Owned property p is transformed to an ALLOY field f_p in the ALLOY signature sig_C. This is possible because all concrete class types and primitive types have a signature definition in the ALLOY model after the flattening step in Section 3.6.1.


LHS: Metamodel
RHS: Generated Alloy Paragraph(s)
Multiple Inheritance Class Hierarchy to Alloy Signatures
SuperClass11 <abstract>
...
SuperClass12 <abstract>
...
SuperClass1M <abstract>
...
SuperClassN1 <abstract>
...
SuperClassN2 <abstract>
...
SuperClassNM <abstract>
...
ConcreteClass1
...
ConcreteClass2
...
ConcreteClassM
...
Alloy Signatures for Multiple Inheritance Hierarchy
Graphical Representation of the Alloy Model
ConcreteClass1 <signature>
...
ConcreteClass2 <signature>
...
ConcreteClassM <signature>
...

```
sig ConcreteClass1
{
...
}

sig ConcreteClass2
{
..
}
...
sig ConcreteClassM
{


}
```

Figure 3.13: Step 1: Flattening the Multiple Inheritance Hierarchy

2. **Inherited Property** *p* **is of Primitive Type in a Concrete Class** *C*:

   Property *p* is transformed to an ALLOY field in the ALLOY signature for *C*. This is possible because all concrete class types and primitive types have a signature definition in the ALLOY model after the flattening step in Section 3.6.1.

3. **Owned Property** *p* **of Concrete Class Type** *CT* **in a Concrete Class** *C*:

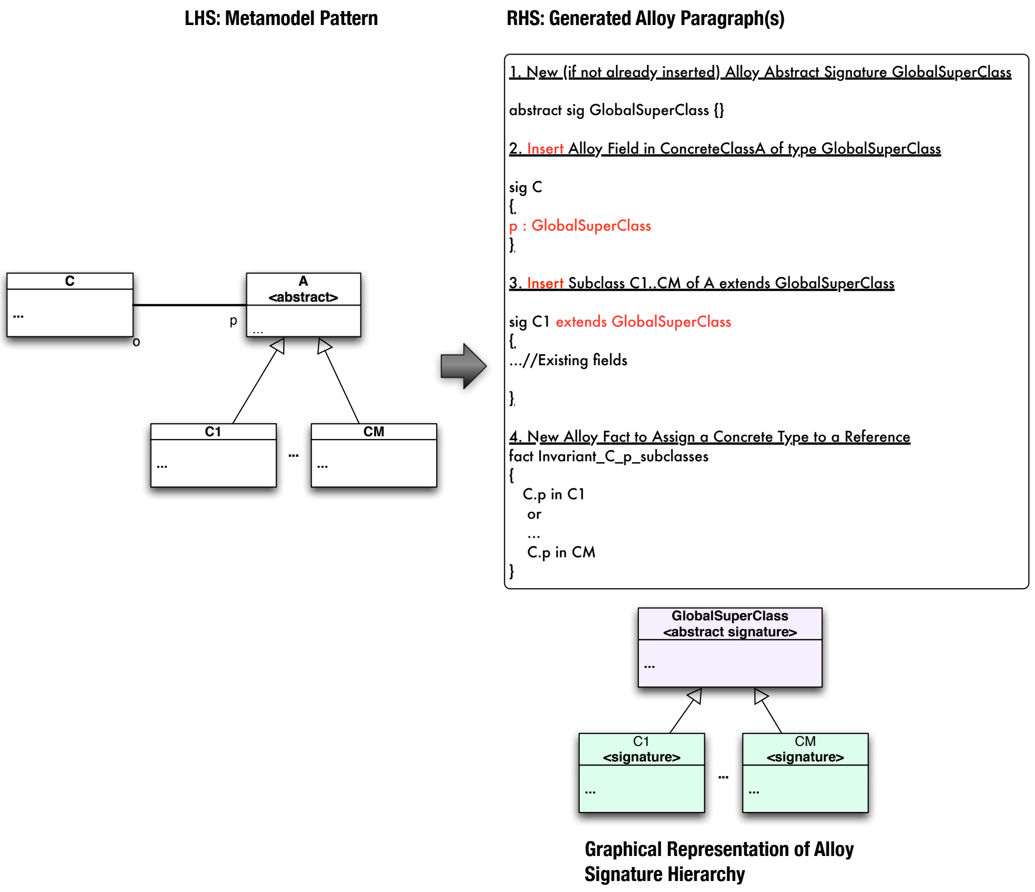
   Owned property *p* of a concrete class type *CT* is transformed into an ALLOY field in the signature representing *C*. The process is identical to the transformation described in Section 3.5.4.

4. **Inherited Property** *p* **of Concrete Class Type** *CT* **in a Concrete Class** *C*: Inherited property *p* of a concrete class type *CT* is transformed into an ALLOY field in the signature representing *C*. The process is identical to the transformation described in Section 3.5.4.

5. **Owned Property** *p* **of Abstract Class Type** *A* **in a Concrete Class** *C*:

   Properties of abstract type cannot be simply transformed to an ALLOY field in a signature for *C*. This is because *A* is not transformed to a signature or an abstract signature in the ALLOY model and hence *p* does not have a type. Therefore, we deal with the transformation in the following steps as shown in Figure 3.14.

   (a) If not already existing we create an abstract signature called GlobalSuperClass and insert it into the ALLOY model. The abstract signature acts as a *placeholder* for abstract classes in the input metamodel.



1. New (if not already inserted) Alloy Abstract Signature GlobalSuperClass

```
abstract sig GlobalSuperClass {}
```

2. Insert Alloy Field in ConcreteClassA of type GlobalSuperClass

```
sig C
{
p : GlobalSuperClass
}
```

3. Insert Subclass C1..CM of A extends GlobalSuperClass

```
sig C1 extends GlobalSuperClass
{
...//Existing fields

}
```

4. New Alloy Fact to Assign a Concrete Type to a Reference

```
fact Invariant_C_p_subclasses
{
   C.p in C1
    or
    ...
    C.p in CM
}
```

Figure 3.14: Transforming Property of Abstract Type to ALLOY fact

(b) We insert the field for *p* into the ALLOY signature for *C* with the type GlobalSuperClass.

(c) All concrete classes *C*1...*CM* that inherit from *A* now inherit from GlobalSuperClass. The inheritance is illustrated on the RHS of Figure 3.14. Concrete classes that do not inherit from an abstract class in the metamodel do not also inherit from GlobalSuperClass.

(d) We generate an ALLOY fact that states that the property *p* of *C* is in one of the concrete subclasses of *A* namely *C*1 , *C*2,..or *CM*. The fact enforces the property to always refer to concrete subclass objects of *C*.

6. **Inherited Property** *p* **of Abstract Class Type** *A* **in a Concrete Class** *C*: Inherited property *p* of a abstract class type *A* is exactly equivalent to transforming an owned property of abstract class type discussed above and illustrated in Figure 3.14. However, we may choose to optimize this transformation.

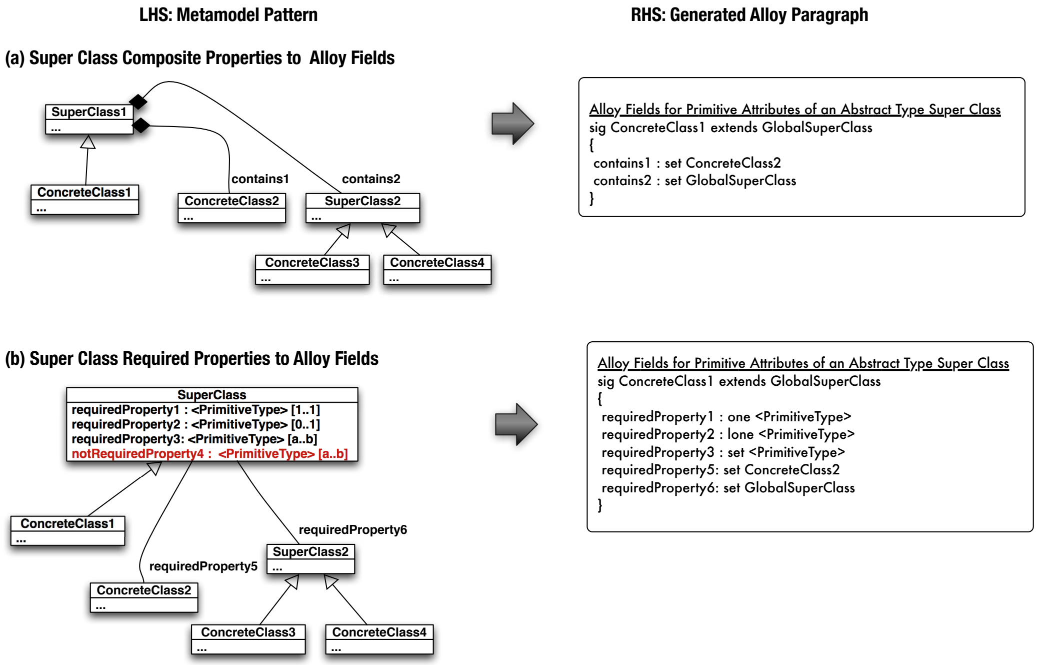
   All inherited properties may be flattened to into a concrete class signature. However, we may also *select properties* (attributes and references) that will be transformed to ALLOY fields. An objective for us is to minimize the number of properties we flatten from the abstract super classes to concrete classes. We use two heuristics. Given an ALLOY signature representing a concrete class,

   (a) We create ALLOY fields only for all inherited properties that can *contain* objects. There properties can contain objects of any of the concrete classes in the metamodel. We perform the transformation to ensure that all objects have a container property (except the top-level container class). This transformation stems from the fact that It is mandatory that objects of all classes have a container in Ecore. Hence, the non-root ALLOY signatures must have a container.

   (b) We create ALLOY fields for all inherited *required properties* for a given application for model generation. For instance, we preserve all properties used by a model transformation for which we intend to generate models. This step helps minimize the size of the constraint satisfaction problem for model generation.

   We illustrate the flattening of composite properties that can contain concrete class objects in Figure 3.15 (a). There are two possibilities while flattening such properties. If a composite property such as *contain1* can hold concrete classes we transform the property as an ALLOY field as shown on the RHS of Figure 3.15 (a). A composite property such as *contain2* may refer to an abstract class that is inherited by several concrete classes. In such a case, we transform the property as an ALLOY field of type GlobalSuperClass. While model generation, the GlobalSuperClass is replaced by objects of concrete subclasses of SuperClass2. This implies that the field ConcreteClass1.contains2 can refer to both objects of type ConcreteClass3 and ConcreteClass4.

   The flattening of required properties is very similar to the flattening of composite properties with the exception that properties that are not required are not transformed as ALLOY


LHS: Metamodel Pattern
RHS: Generated Alloy Paragraph
(a) Super Class Composite Properties to Alloy Fields
SuperClass1
...
ConcreteClass1
...
contains1
contains2
ConcreteClass2
...
SuperClass2
...
ConcreteClass3
...
ConcreteClass4
...
(b) Super Class Required Properties to Alloy Fields
SuperClass
requiredProperty1 : <PrimitiveType> [1..1]
requiredProperty2 : <PrimitiveType> [0..1]
requiredProperty3: <PrimitiveType> [a..b]
notRequiredProperty4 : <PrimitiveType> [a..b]
ConcreteClass1
...
requiredProperty6
requiredProperty5
SuperClass2
...
ConcreteClass2
...
ConcreteClass3
...
ConcreteClass4
...

```
Alloy Fields for Primitive Attributes of an Abstract Type Super Class
sig ConcreteClass1 extends GlobalSuperClass
{
 contains1 : set ConcreteClass2
 contains2 : set GlobalSuperClass
}
```

```
Alloy Fields for Primitive Attributes of an Abstract Type Super Class
sig ConcreteClass1 extends GlobalSuperClass
{
 requiredProperty1 : one <PrimitiveType>
 requiredProperty2 : lone <PrimitiveType>
 requiredProperty3 : set <PrimitiveType>
 requiredProperty5: set ConcreteClass2
 requiredProperty6: set GlobalSuperClass
}
```

Figure 3.15: Flattening Properties in the Multiple Inheritance Hierarchy

fields. For instance, in Figure 3.15 (b) the primitive type property notRequiredProperty4 is not transformed to an ALLOY field.

3.6.3 Transforming Implicit Constraints to ALLOY Facts

In the third step, we transform implicit constraints in a metamodel with multiple inheritance to ALLOY. We present the transformations as follows:

Transforming Opposite Properties to ALLOY Facts

First, we consider the transformation of opposite properties to ALLOY facts. We recall that an opposite property represents a bi-directional relationship between two classes. After the property flattening process, an opposite property in a concrete class may refer to an abstract class or a concrete class. An opposite property between two concrete classes can be transformed using the rule already presented in Section 3.5.5. However, an opposite property between a concrete class and an abstract class leads to the generation of a different ALLOY fact since all abstract classes are not included in the ALLOY model. We illustrate this transformation in Figure 3.16.

The transformation states that if any property propertyB of concrete class ConcreteClassA has an opposite property propertyA in the abstract super class SuperClassB then,

1. If ConcreteClassA.propertyB refers to any object object2 of ConcreteClass1 then ConcreteClass1.propertyA refers to ConcreteClassA.

2. If ConcreteClassA.propertyB refers to any object object2 of ConcreteClass2 then ConcreteClass2.propertyA refers to ConcreteClassA.

3. ...

4. If ConcreteClassA.propertyB refers to any object object2 of ConcreteClassN then ConcreteClassN.propertyA refers to ConcreteClassA.

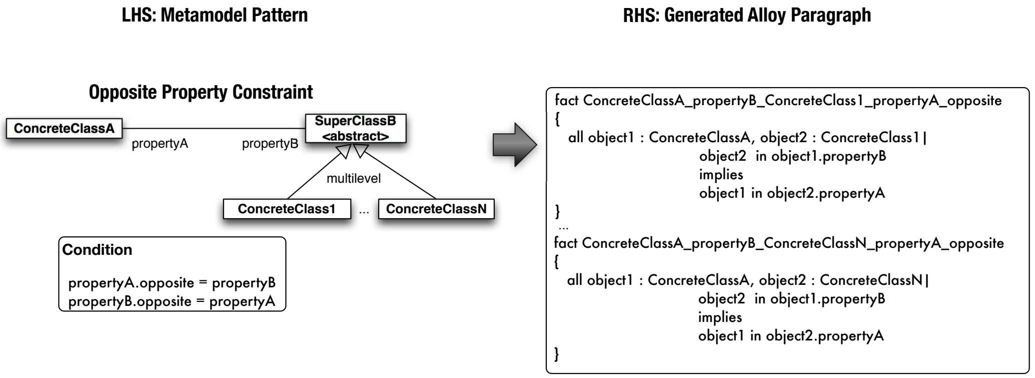
We generate facts for opposite properties to all possible sub-classes of the abstract super class SuperClassB.

Transforming Composite Properties to ALLOY Facts

A composite property for each concrete class in the metamodel is transformed to an ALLOY fact. The transformation is identical to the transformation we have already seen for composite properties in a metamodel with single inheritance. See Section 3.5.5 for more detail.

Transforming Containers of a Class to ALLOY Facts

A concrete class can be contained by another concrete class or an abstract super class as shown in Figure 3.17. Therefore, any object of ConcreteClassA can either be contained by the concrete class ConcreteClassB or all subclasses of abstract SuperClasB such as ConcreteClass1,...,ConcreteClassN.



```
fact ConcreteClassA_propertyB_ConcreteClass1_propertyA_opposite
{
   all object1 : ConcreteClassA, object2 : ConcreteClass1 |
                              object2  in object1.propertyB
                              implies
                              object1 in object2.propertyA
}
 ...
fact ConcreteClassA_propertyB_ConcreteClassN_propertyA_opposite
{
   all object1 : ConcreteClassA, object2 : ConcreteClassN |
                              object2  in object1.propertyB
                              implies
                              object1 in object2.propertyA
}
```

Figure 3.16: Transforming Opposite Properties to ALLOY Facts in Metamodel with Multiple Inheritance

The Alloy fact on the RHS of the transformation in Figure 3.17 depicts the containment constraint. All objects of ConcreteClassA will be contained by either ConcreteClassB, ConcreteClass1,..,or ConcreteClassN. The fact also states that an object of ConcreteClassA can have only one containing object.

Transforming Multiplicity Constraints of a Class to ALLOY Facts

Multiplicity constraints on properties for each concrete class in the metamodel is transformed to an ALLOY fact. The transformation is identical to the transformation we have already seen for multiple properties in a metamodel with single inheritance. See Section 3.5.5 for more detail.

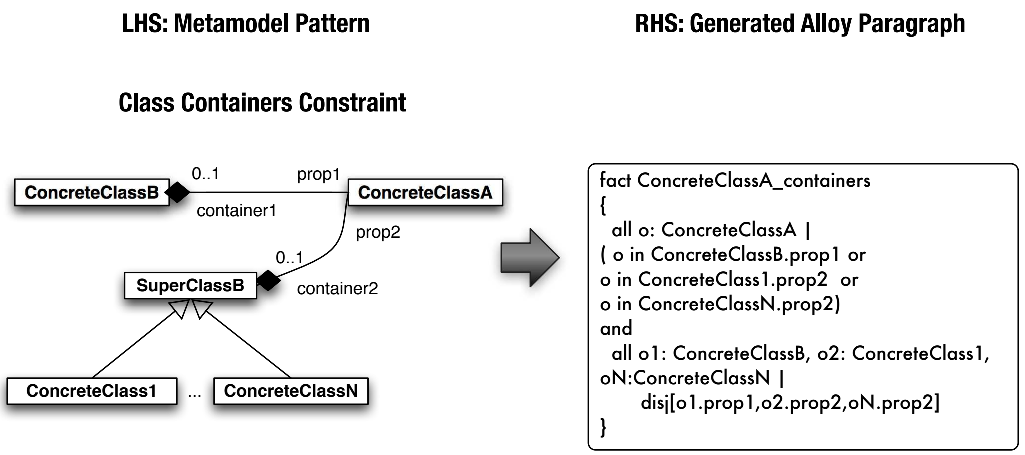
Transforming Identity Properties of a Class to ALLOY Facts

Identity properties on properties for each concrete class in the metamodel is transformed to an ALLOY fact. The transformation is identical to the transformation we have already seen for identity properties in a metamodel with single inheritance. See Section 3.5.5 for more detail.

3.7 Handling the Transformation of Metamodel Invariants to ALLOY Facts

Metamodel invariants are textual constraints on a metamodel. We express some constraints textually due to limitations of class diagrams/Ecore model in describing constraints on the modelling domain. Textual constraints are often specified using the industry standard language OCL. An OCL constraint is specified on a pattern of a model in a modelling language. For instance, the constraint that *no cyclic inheritance* can exist in an UML class diagram can be represented in OCL as shown in Listing 3.1.

```
context Class
```



```
fact ConcreteClassA_containers
{
  all o: ConcreteClassA |
( o in ConcreteClassB.prop1 or
o in ConcreteClass1.prop2  or
o in ConcreteClassN.prop2)
and
  all o1: ConcreteClassB, o2: ConcreteClass1,
oN:ConcreteClassN |
      disj[o1.prop1,o2.prop2,oN.prop2]
}
```

Figure 3.17: Transforming Containers of a Concrete Class to an ALLOY Fact in Metamodel with Multiple Inheritance

```
  inv noCyclicInheritance
    not self.allGenerals()->includes(self)
```

Listing 3.1: An Example OCL Constraint

Automating the transformation of all OCL constraints to ALLOY facts is not within the scope of this thesis. We manually transform all OCL constraints in this thesis to ALLOY facts. Developers experienced in both OCL and ALLOY can extract the meaning of an OCL constraint and express it as an ALLOY fact.

For example, we transform the constraint in Listing 3.1 to the ALLOY fact in Listing 3.2.

```
fact noCyclicInheritance
{
  no c: Class | c in c.^general
}
```

Listing 3.2: ALLOY fact representing No Cyclic Inheritance

The ALLOY fact uses *transitive closure* to enforce the constraint that no Class *c* exists such that it is contained in either *c.general or c.general.general or c.general.general.general,...* and so on. This implies that no multi-level super classes of a class can contain it thereby eliminating the cyclic inheritance in all UML class diagram models. During the course of the thesis a number of OCL constraints have been manually transformed to ALLOY.

3.8 Illustration of Transformation to ALLOY

We transform the effective metamodel of UML with 26 classes and 65 properties, shown in Figure 3.5 to an ALLOY model. The resulting ALLOY model contains signatures only for the *concrete classes* in the metamodel.

The generated signatures are shown in Listing 3.3.

```
 module EffectiveUML

 open util/boolean as Boolean

 sig GlobalSuperClass { }

one sig Package extends GlobalSuperClass
 {

  packagedElement:set GlobalSuperClass, //PackageableElement

  name:one Int
 }
sig Association extends GlobalSuperClass
 {

  ownedEnd:set Property,

  memberEnd:set Property,

  attribute:set Property,

  name:one Int
 }
sig Property extends GlobalSuperClass
 {

  datatype:one DataType,

  owningAssociation:one Association,

  association:one Association,

  name:one Int
 }
sig Class extends GlobalSuperClass
 {

  nestedClassifier:set GlobalSuperClass, //Classifier

  ownedAttribute:set Property,

  attribute:set Property,

  name:one Int
 }
sig DataType extends GlobalSuperClass
 {

  ownedAttribute:set Property,

  attribute:set Property,

  name:one Int
 }
sig PrimitiveType extends GlobalSuperClass
 {

  ownedAttribute:set Property,

  attribute:set Property,

  name:one Int
 }
```

Listing 3.3: Generated ALLOY Signatures in Effective UML

Signatures for concrete classes (those that initially inherit from abstract superclasses) now extend the abstract signature GlobalSuperClass in the ALLOY model. For instance, Association inherits from PackageableElement in UML. Therefore, the signature Association extends GlobalSuperClass.

Each property (owned/inherited/primitive) of a concrete class *C* is transformed to an ALLOY field in the signature representing *C*. For instance, in Listing 3.3, the primitive property *name* in Association is an inherited property from NamedElement that is directly transformed to a field in the Association signature. Similarly, the property *ownedEnd* of Association is of a concrete class type Property. The property is directly transformed to an ALLOY field in the Association signature. A property may have an abstract class type in the metamodel. For instance, the property *nestedClassifier* of Class is of abstract class type Classifier. There is no signature for Classifier in the ALLOY model. Therefore, the property is transformed to an ALLOY field of type GlobalSuperClass in the ALLOY model. We generate ALLOY facts for fields of type GlobalSuperClass. These ALLOY facts state that the type of the field is one or more of the signatures already in the ALLOY model. In fact these signatures represent the exact concrete subclasses of the abstract class type. For instance, in Listing 3.4 we present two such generated facts. The second fact states that all objects of type Class.nestedClassifier must be of type Class or DataType, or PrimitiveType. Class, DataType, and PrimitiveType are concrete classes that inherit from Classifier in the UML metamodel.

```
 fact  Invariant_Package_packagedElement_subclasses
 {
Package.packagedElement in Package or Package.packagedElement in Association or Package.packagedElement in Class or
       Package.packagedElement in DataType or Package.packagedElement in PrimitiveType
 }

 fact  Invariant_Class_nestedClassifier_subclasses
 {
Class.nestedClassifier in Association or Class.nestedClassifier in Class or Class.nestedClassifier in DataType or
       Class.nestedClassifier in PrimitiveType
 }
```

Listing 3.4: Generated ALLOY Facts for Subclasses in Effective UML

We generate ALLOY facts for opposite properties, composite properties, and containers in the effective UML metamodels. We present examples of these facts in Listing 3.5.

The first fact in Listing 3.5 enforces the *opposite property constraint* between two properties Association.ownedEnd and Property.owningAssociation. The fact states that if any Property object is in the set o.ownedEnd (where o is an Association object) then o is in the set o1.owningAssociation (where o1 is a Property object).

The second fact enforces the *composite property constraint* for the property Package.packagedElement. The fact states that for each object o1, o2 of type Package, and for each property p1 in o1.packagedElement, p2 in o2.packagedElement, if p1 is equal to p2 then the containing objects o1 and o2 are one and the same. The constraint enforces that if packagedElement refers to an object of type PackagedElement then the object can have exactly one Package container.

The third fact enforces the *containers constraint* for the class Association. Association objects can be contained by two different containers namely Package.packagedElement and Class.nestedClassifier. The fact first states that each Association object o is either contained by Package.packagedElement or Class.nestedClassifier. Second, the fact states that objects contained by Package.packagedElement cannot be contained by Class.nestedClassifier and vice versa.

```
//1. An Example of a fact generated for Opposite Property of Association.ownedEnd and Property.owningAssociation


 fact  Invariant_Association_ownedEnd_Property_owningAssociation_opposite
 {

 all o :Association, o1:Property |
   (o1 in o.ownedEnd implies o in o1.owningAssociation)

 }
```

```
//2. An Example of a fact generated for Composite Property Package.packagedElement


 fact  Invariant_Package_packagedElement_composite
 {

 all o1 :Package, o2:Package |
   all p1:o1.packagedElement,p2:o2.packagedElement|p1=p2 implies o1=o2
 }


//3. An Example of a fact generated for Containers of Association Objects

fact Invariant_Association_containers
{
 all o : Association | (o in Package.packagedElement or
o in Class.nestedClassifier )
 and

 all o1 : Package,o2 : Class | disj[o1.packagedElement,o2.nestedClassifier]
}
```

Listing 3.5: Generated ALLOY Facts for Implicit Constraints in Effective UML

The entire solvable ALLOY model for the effective metamodel is available for download at this site [4].

3.9 Model Generation by Solving ALLOY Model

As a consequence of the transformation steps described in the previous sections we obtain the ALLOY model of an effective modelling domain. The ALLOY model contains a set of set signatures representing the concepts and their relationships in a domain. It also contains a set of facts that encode implicit constraints in a metamodel. In this section, we demonstrate how we can generate models in the effective modelling domain specified as a constraint satisfaction problem in ALLOY. The generation of models in ALLOY must satisfy an ALLOY predicate (which may subsume other predicates). Objective-specific knowledge such as for test model generation may help specify such predicates or an empty predicate representing no new knowledge. In Section 3.9.1, we introduce specification of ALLOY predicates to guide generation of models in an effective modelling domain. In ALLOY solving for a predicate implies generation of models that satisfy the predicate and all ALLOY facts. ALLOY allows generation of models within a certain scope or within finite-bounds on the number of objects for each type. Therefore, in Section 3.9.2 we describe the specification to guide generation of models in a finite scope.

3.9.1 Specifying ALLOY Predicates to Guide Generation

Empty Alloy Predicate

If the goal is to generate models in the modelling domain specified only by the metamodel and the invariants we do not need to guide generation with more information. Therefore, we generate an empty ALLOY predicate as shown in Listing 3.6.

```
pred Unguided
{

}
```

Listing 3.6: Empty ALLOY Predicate

MF1{Classifier(name=””) and Classifier(name=”.+”)}
MF2{Class(is_persistent = true) and Class(is_persistent = false)}
MF3{Class(parent = 0) and Class(parent = 1)}
MF4{Class(ownedAttribute = 1)and Class(ownedAttribute > 1)}
MF5{Attribute(is_primary = true) and Attribute(is_primary = false)}
MF6{Attribute(name=””) and Attribute (name=”.+”)}
MF7{Attribute(type=0) and Attribute (type=1)}
MF8{Association (name=””) and Association (name=”.+”)}
MF9{Association(ownedEnd=0) and Association(ownedEnd=1)}
MF10{Association(memberEnd=0) and Association(memberEnd =1)}
MF11{Package(packagedElement=1) and Package(packagedElement>1)}

Figure 3.18: Some Model Fragments from effective UML metamodel

Objective-specific ALLOY Predicates

A number of objective-specific sources of knowledge may lead to generation or specification of ALLOY predicates to guide generation with an objective. We explain the generation of such predicates with the help of two examples.

In the first example, an objective-specific source of knowledge may be the *pre-condition* of model transformation. We consider the model transformation from UML class diagrams to Relational Database Management Systems (RDBMS) models called class2rdbms. For instance, the pre-condition the transformation states that all classes in the input model must have at least one primary attribute. The condition is necessary for indexing and may be expressed in the predicate shown in Listing 3.7.

```
//All Classes must contain at least one primary attribute
pred atleastOnePrimaryAttribute
{
  all c:Class| some a:c.attrs |  a.is_primary==True
}
```

Listing 3.7: A model transformation pre-condition in ALLOY

The property *isPrimary* of the class Attribute is not part of the original UML specification. It has been added to the effective UML metamodel as a new property of the class *Class*. Similarly, we add the property *is_persistent* to the class Class to enable serialization of classes to RDBMS models.

In the second example we use knowledge based on input domain partitioning to guide model generation. Input domain partitioning [153] is a well-known source of knowledge to ensure coverage of the input domain for software testing. Partitions of the modelling domain or the metamodel are a source of knowledge to generate ALLOY predicates. These ALLOY predicates ensure that the entire modelling domain is covered. In previous work, Franck et. al. [55] extract partitions of an input metamodel known as *model fragments*. For instance, the following model fragment states that the model to be generated must contain at least one "Classifier" object with an empty name attribute and a "Classifier" object with non-empty name.

Classifier(*name* =“ ”) and Classifier(*name* =“.+”)

The model fragment can be transformed to an ALLOY predicate as shown in Listing 3.8.

```
pred modelFragment
{
 some c1:Classifier, c2:Classifier| c1.name=0 and c2.name!=0
}
```

Listing 3.8: Model Fragment ALLOY Predicate

In Figure 3.18, we present some of the important model fragments generated from the effective UML metamodel.

3.9.2 Specifying ALLOY Run Commands with Finite Bounds

A *run command* tells ALLOY to search for an instance of a predicate. We may specify a scope that bounds the size of the instances of the ALLOY model. The basic run command in shown in Listing 3.9. The command attempts to generate an instance that satisfies the predicate *example* in the finite scope of 20.

```

//A Basic Run Command
pred example() {}

run example for  20
```

Listing 3.9: Basic ALLOY Run Command

We can go a step further and control the generation of models with variable scope for each signature. The scope for integer and sequences may be specified as well. For instance, a scope of *5 int* implies an instance can contain integers between $-2^5 and 2^5$. Similarly, *5 seq* implies that an instance can contain sequences up to a size of 5. The Listing 3.10 illustrates a run command with variable scope.

```

//A Variable Scope Run Command
pred example() {}

run example for 1 Package, 5 Class, 5 Association, 3 PrimitiveDataType,...5 int, 5 seq
```

Listing 3.10: ALLOY Run Command with Variable Scope

If known in advance, we may also specify the exact scope for a signature as shown in Listing 3.11.

```

//A Variable Scope Run Command
pred example() {}

run example for 1 Package, exactly 5 Class, 5 Association, exactly  3 PrimitiveDataType,...5 int, 5 seq
```

Listing 3.11: ALLOY Run Command with Exact Scopes

3.9.3 ALLOY Instances to EMF models

The ALLOY instances generated are in the form of atoms and relations between atoms. They need to be transformed back to models that conform to a metamodel. This transformation is rather straightforward as the ALLOY instances have a structure very similar to objects with properties. CARTIER contains a Java module that traverses the ALLOY instance and instantiates objects with properties of the input Ecore metamodel. The ALLOY instance acts as a source of information to recreate a valid model of the input metamodel.

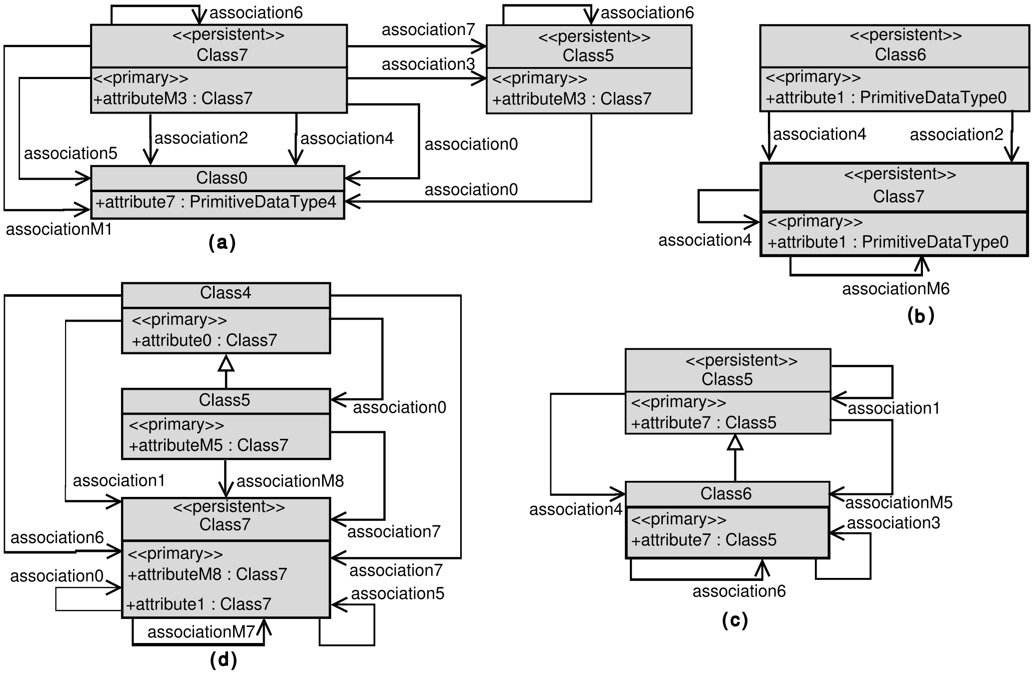


Figure 3.19: (a) Model conforming to Meta-model (b)Model conforming to Meta-model + Pre-condition (c) Model conforming to Meta-model+ Pre-condition + Test Model Objective (d) Model conforming to Meta-model + Pre-condition + Model Fragment

3.10 Illustrative Examples: Generation UML Class Diagram Models

We generate models from the input domain of the class2rdbms transformation using the different sources knowledge discussed in Section 3.9.1. We show the selection of 4 UML Class Diagram (UMLCD) models.

To begin, we use the ALLOY analyzer to generate a model that conforms only to the effective UMLCD meta-model. This is shown in Figure 3.19 (a) using UMLCD concrete syntax. The selected test model was found in a *scope* of 10. The scope is the maximum number of objects for each type (or class) in the meta-model. The model selection is performed up to the limit proposed by the scope. We see that the resulting model satisfies all meta-model constraints. However, an attribute of Class0 is not primary. This implies that it is not a valid input to class2rdbms.

The second generated model must contain Class objects with at least one primary attribute which is a pre-condition for transforming UML class diagrams to indexable RDBMS models. The model is shown in Figure 3.19 (b). The selected model has classes with at least one primary attribute just as required by the pre-condition. The selected model was found in a maximum scope of 20. We note that the model now has two classes Class6 and Class7, both of which have at least one primary attribute.

Sources of Knowledge	**Time(sec)**
Meta-model Only	0.78
Meta-model + Pre-condition	7.813
Meta-model + Pre-condition + Test Model Objective	7.97
Meta-model + Pre-condition + Model Fragments	10.477

Table 3.2: Test Model Selection Times

Third, we generate a model that has some classes with $is\_persistent = True$ which is transformation test specific objective. We generate a model in a maximum scope of 20. The resulting model is shown in Figure 3.19 (c). We note the class Class5 is persistent as per the objective. RDBMS models generated from such an input model is

Finally, we introduce model fragment facts along with the meta-model and pre-condition. The model that covers the meta-model and 5 model fragments is shown in Figure 3.19 (d). The resulting model covers some of the model fragments facts we generated from the Ecore model. The model is selected for a maximum scope of 20. The model fragments covered ,as described in Figure 3.18, were MF2, MF3, MF4, MF5. This guarantees that the equivalence classes for property values are covered at least once by a test model. In terms of test qualification, this increases the trust we have in the test models, based on input domain coverage.

In Table 3.2, we summarize the time taken (on a P4 2.6Ghz desktop, with 1Gb RAM) to generate models. From the table we can generally say that more knowledge we have the longer it takes to generate models.

3.11 Validity and Complexity of Transformation to ALLOY

We need to validate the transformation from a metamodel and its invariants to an ALLOY model. Therefore, we ask the question: *Are all solutions of the* ALLOY *model in the modelling domain specified by the metamodel and constraints from heterogenous sources?*. We may answer this by generating all possible solutions of the ALLOY model in a finite scope and checking if each model conforms to the metamodel. However, generating all possible models is computationally expensive. Therefore, can be generate an effective subset of all possible models? In Chapter 4, we perform model generation experiments that cover the modelling domain using partitioning strategies. We demonstrate that all effectives models generated conform to the input metamodel.

The transformation from an effective modelling domain to ALLOY has *linear time complexity*. The transformation involves 2 passes for transforming the metamodel and 1 pass for transforming implicit constraints in a metamodel such as composite properties, opposite properties, etc. to ALLOY facts. Therefore, the time complexity is $O(3*N)$ where N is the number of concepts (total number of classes and properties) in the input metamodel.

3.12 Summary

In this chapter we present three important steps in automatic model discovery. The first step is the metamodel pruning algorithm which is used to obtain the effective metamodel given an input metamodel. We illustrate pruning on UML, a very large input metamodel, to obtain an effective metamodel that represents the class diagram subset of UML. The second step is the transformation of heterogeneous sources of knowledge including the effective metamodel, metamodel invariants, partial model and possibly several domain-specific sources to a common constraint model in ALLOY. We demonstrate the transformation of the class diagram subset of UML and other sources of knowledge such as a simple partial model and model fragments for test models to ALLOY. In the third and the final step we illustrate the generation of models that conform to various sources of knowledge. In particular, we illustrate test model generation and partial model completion for UML class diagrams. In the next chapter, we present experiments illustrating the application of automatic model discovery.

Chapter 4

Experiments in Effective Model Discovery

In this chapter, we present two domain-specific experiments that apply and validate automatic effective model discovery already described in Chapter 3.

1. The first application is to synthesize thousands of models to test a model transformation using testing specific knowledge known as *input domain coverage criteria*|. We qualify the effectiveness or bug detecting ability of these models via *mutation analysis* [107].

2. The second application is to generate model completion recommendations for a *partial model*. The partial model is specified in a domain-specific model editor.

The chapter is organized as follows. In Section 4.1, we describe the model transformation testing application. We present model discovery as model completion in a model editor in Section 4.2.

4.1 Automatic Model Synthesis for Model Transformation Testing

Model transformations are core MDE components that automate important steps in software development such as refinement of an input model, re-factoring to improve maintainability or readability of the input model, aspect weaving, exogenous and endogenous transformations of models, and generation of code from models. Although there is wide spread development of model transformations in academia and industry the validation of transformations remains a hard problem [19]. In this study, we address the challenges in validating model transformations via *black-box automatic test data generation*. We think that black-box testing is an effective approach to validating transformations due to the diversity of transformation languages based on graph rewriting [17] (AToM^3 [67]), imperative execution (Kermeta [108]), and rule-based transformation (ATL [75]) that render language specific formal methods and white-box testing currently impractical.

In black-box testing of model transformations we require *test models* that can *detect bugs* in the model transformation. These models are graphs of inter-connected objects that must conform to a meta-model and satisfy meta-constraints such as well-formedness rules, transformation pre-conditions, and test strategies. Manually specifying several hundred test models targeting various testing objectives is a tedious task and in many cases impossible since the modeller may have to simultaneously satisfy numerous possibly inter-related constraints.

In this section, we apply our automatic model discovery framework CARTIER previously discussed in Chapter 3 to *automatic test model generation*. CARTIER has to address two main problems for test generation: identify a precise model of the transformation's input domain; automatically select relevant test models in the input domain. The first issue is related to the fact that the input domain of a transformation is generally described with a general purpose metamodel (e.g., UML). However, the effective input domain, that captures only the set of models that can be transformed, is much smaller than the set of instances of the general purpose metamodel. CARTIER can prune the metamodel in order to explicitly build a subset of the metamodel that the transformation can manipulate. CARTIER also assists the definition of pre conditions on the metamodel to make the input domain more precise. Once the input domain is precisely modelled, CARTIER can generate models in the input domain. CARTIER either generates models without guidance or it can use test strategies in order to have models that cover the input domain [55].

Are the test models generated by CARTIER able to detect bugs in a model transformation? We answer this question by generating and comparing sets of test models using different testing strategies. Specifically, we consider two testing strategies: *unguided* and *input domain coverage strategies* [55]. We use *mutation analysis* [49] [107] for model transformations to compare these testing strategies. Mutation analysis serves as a *test oracle* to determine the relatively adequacy of generated test sets.

We perform experiments to generate test models using different testing strategies and qualify them using mutation analysis. We generate test models for the representative model transformation of Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms. The mutation scores show that input domain coverage strategies guide model generation with considerably higher bug detection abilities (93%) compared to unguided generation (70%). These results are based on 3200 generated test models and several hours of computation on a 10 machine grid of high-end servers. The large difference in mutation scores between coverage strategies and unguided generation can be attributed to the fact that coverage strategies enforce several aspects on test models that unguided generation fail to do. For instance, coverage strategies enforce injection of *inheritance* in the UMLCD test models. Unguided strategies do not enforce such a requirement. Several mutants are killed due to test models containing inheritance.

The *scientific contribution* in this section addresses three important questions:

- **Question 1:** How can we scale the approach to generating test models for large input meta-models such the UML?

- **Question 2:** Does the model transformation pre-condition precisely specify the input domain of a model transformation? If not, can automatically generated test models help improve the pre-condition by presenting unforeseen and unwanted modelling patterns?

- **Question 3:** Are we consistently able to generate effective test models for a given strategy using our approach?

The precise contributions of this section addresses exactly these problems. We enlist them below:

- **Contribution 1:** We use *meta-model pruning* (see Chapter 3, Section 3.4, [141]) to prune a large input meta-model such as the UML to a subset called the effective input meta-model. The effective input meta-model contains only classes, properties, their dependencies relevant the transformation under test. The often smaller effective input meta-model is transformed to a small formal representation in ALLOY. In contrast, transforming a large input meta-model such as the whole of UML to ALLOY results in a formal model that renders SAT solving intractable due to the large number of signatures and facts.

- **Contribution 2:** We show how automatically generated test models can help us improve a model transformation's pre-condition. For instance, the test models we generate for the case study transformation class2rdbms helps us discover new pre-condition constraints. These pre-conditions were not initially envisaged by the panel of world experts in model-driven engineering who propose the class2rdbms as the benchmark case study at the MTIP workshop [22]. We show that automatic generation can help us rapidly discover structures that human or even experts cannot preview in advance or require several years of transformation usage experience.

- **Contribution 3:** We show that CARTIER consistently generates effective test models for a given strategy. We illustrate consistency by demonstrating that generating multiple test models for the same test strategy does not significantly change mutation scores. These test models correspond to multiple non-isomorphic solutions obtained using ALLOY's symmetry breaking scheme [143].

4.1.1 Problem Description

We present the problem of black-box testing *model transformations*. A model transformation $MT(I,O)$ is a program applied on a set of input models I to produce a set of output models O as illustrated in Figure 4.1. The set of all input models is specified by a meta-model MM_I. The set of all output models is specified by meta-model MM_O. The pre-condition of the model transformation $pre(MT)$ further constrains the input domain. A post-condition $post(MT)$ limits the model transformation to producing a subset of all possible output models. The model transformation is developed based on a set of requirements $MT_{Requirements}$.

Model generation for black-box testing involves finding valid input models we call *test models* from the set of all input models I. Test models must satisfy constraints that increase the trust in the quality of these models as test data and thus should increase their capabilities to detect bugs in the model transformation $MT(I,O)$. Bugs may also exist in the input meta-model and its invariants MM_I or the transformation pre-condition $pre(MT)$. However, in this study we only focus on detecting bugs in a transformation.

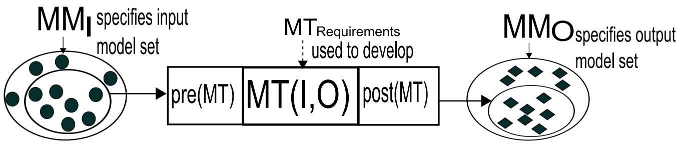

MMI specifies input model set
MTRequirements used to develop
MMO specifies output model set
pre(MT)
MT(I,O)
post(MT)

Figure 4.1: A Model Transformation

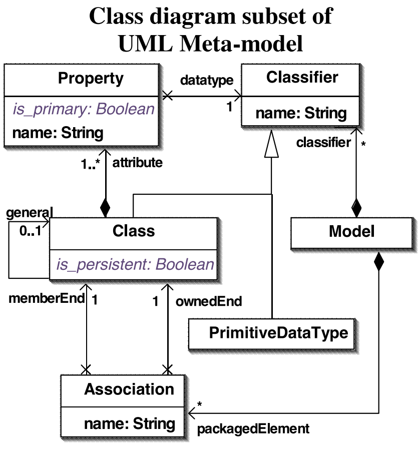
4.1.2 Transformation Case Study

Our case study is the transformation from UML Class Diagram models to RDBMS models called class2rdbms. In this section we briefly describe class2rdbms and discuss why it is a representative transformation to validate test model generation strategies.

In black-box testing we need input models that conform to the input meta-model MM_I and transformation pre-condition $pre(MT)$. Therefore, we only discuss the MM_I and $pre(MT)$ for class2rdbms and avoid discussion of the model transformation output domain. In Figure 4.2, we present a subset of the UML input meta-model for class2rdbms. The concepts and relationships in the input meta-model are stored as an Ecore model [58] (Figure 4.2 (a)). The invariants on the UMLCD Ecore model, expressed in Object Constraint Language (OCL) [114], are shown in Figure 4.2 (b). The Ecore model and the invariants together represent the true input meta-model for class2rdbms. The OCL and Ecore are industry standards used to develop meta-models and specify different invariants on them. OCL is not a domain-specific language to specify invariants. However, it is designed to formally encode natural language requirements specifications independent of its domain. In [155] the authors present some limitations of OCL.

The input meta-model MM_I gives an initial specification of the input domain. However, the model transformation itself has a pre-condition $pre(MT)$ that test models need to satisfy to be correctly processed. Constraints in the pre-condition for class2rdbms include: (a) All Class objects must have at least one primary Property object (b) The type of an Property object can be a Class C, but finally the transitive closure of the type of Property objects of Class C must end with type PrimitiveDataType. In our case we approximate this recursive closure constraint by stating that Property object can be of type Class up to a depth of 3 and the 4th time it should have a type PrimitiveDataType. This is a finitization operation to avoid navigation in an infinite loop. (c) A Class object cannot have an Association and an Property object of the same name (d) There are no cycles between non-persistent Class objects. These initial pre-conditions are transformed to ALLOY and are presented in Appendix 6.5.

We choose class2rdbms as our representative case study to validate input selection strategies. It serves as a sufficient case study for several reasons. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [22] to experiment and validate model transformation language features. The input domain meta-model of UML class diagram model covers all major meta-modelling concepts such as inheritance, composition, finite and infinite multiplicities. The entire UML input meta-model serves as a large input meta-model to demonstrate meta-model pruning to an effective input meta-model containing only class diagram concepts.The constraints on the UML meta-model contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model



(a)

Samples of **OCL Invariants**

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

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

(b)

Figure 4.2: (a) Class Diagram Subset of UML Ecore Meta-model (b) OCL constraints on the Ecore meta-model

such as there must be no cyclic inheritance. The class2rdbms exercises most major model transformation operators such as navigation, creation, and filtering (described in more detail in [107]) enabling us to test essential model transformation features. Among the limitations the UMLCD meta-model does not contain Integer and Float attributes. The number of classes in the UMLCD meta-model is not very high when compared to the standard UML 2.0 specification. There are also no inter meta-model references and arbitrary containments in the simple meta-model. However, this not really limitation in our approach as we claim that specifying a test model requires only a small subset of the entire meta-model and extracting this subset via meta-model pruning is part of our methodology.

Model generation is relatively fast but performing mutation analysis is extremely time consuming. Therefore, we perform mutation analysis on class2rdbms to qualify transformation and meta-model independent strategies for model synthesis. If these strategies prove to be useful in the case of class2rdbms then we recommend the use of these strategies to guide model synthesis in the input domain of other model transformations as an initial test generation step. For instance, in our experiments, we see that generation of a 15 class UMLCD models takes about 20 seconds and mutation analysis of a set of 20 such models takes about 3 hours on a multi-core high-end server. Generating thousands of models for different transformations takes about 10% of the time while performing mutation analysis takes most of the time.

4.1.3 Automatic Test Model Generation and Qualification Methodology

We outline the methodology for test generation using CARTIER and qualification of the generated test models via mutation analysis in Figure 4.3. Concisely, the test model generation methodology follows the steps:

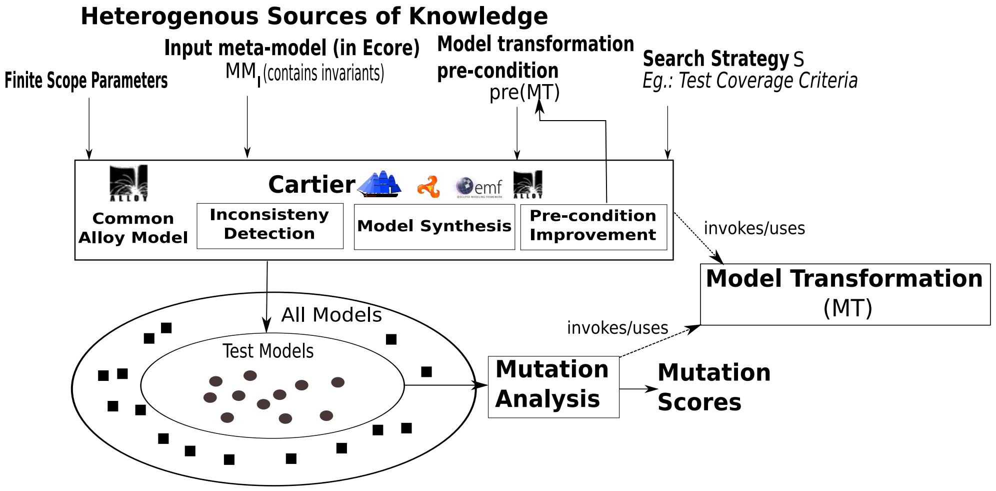

Heterogenous Sources of Knowledge
Finite Scope Parameters
Input meta-model (in Ecore)
MM_I (contains invariants)
Model transformation pre-condition
pre(MT)
Search Strategy S
Eg.: Test Coverage Criteria
Cartier
emf
Common Alloy Model
Inconsisteny Detection
Model Synthesis
Pre-condition Improvement
invokes/uses
Model Transformation (MT)
invokes/uses
All Models
Test Models
Mutation Analysis
Mutation Scores

Figure 4.3: CARTIER Methodology for Automatic Test Generation and Mutation Analysis based Qualification

1. CARTIER performs static analysis on the model transformation MT to obtain the initial set of used types and properties.

2. CARTIER performs metamodel pruning of MM_I using these used types and properties to obtain the effective input metamodel eMM_I (details in Chapter 3, Section 3.4)

3. CARTIER transforms eMM_I, its invariants C, the transformation pre-condition $pre(MT)$ and test strategy to an ALLOY model (details in Chapter 3, Sections 3.5, 3.6, 3.7).

4. CARTIER generates models to detect inconsistencies in test strategy predicates. These test strategy predicates in ALLOY are automatically generated in the previous step and are included in the ALLOY model. For instance, a predicate contains a model fragment that is desirable in a test model (see Section 3.9 for more information on fragments). We attempt to synthesize a model that satisfies the conjunction of the predicate, the ALLOY model representation of the metamodel eMM_I, its invariants C, and $pre(MT)$. If we fail to generate a model in a maximum finite scope then we eliminate the predicate as it is inconsistent with eMM_I, its invariants C, and $pre(MT)$ (introduced in Section 3.9)

5. Finally, CARTIER generates sets of test models that satisfy all consistent predicates representing test strategies in a finite scope using run commands for each predicate (introduced in Chapter 3, Section 3.9). It can also generate multiple non-isomorphic test models by soliciting ALLOY's symmetry breaking scheme [143] currently applicable to the MiniSAT [51] SAT solver.

The generated models may lead to raising of general exceptions such as memory leaks, divide by zero, infinite loops in the model transformation MT as its initial pre-condition definition

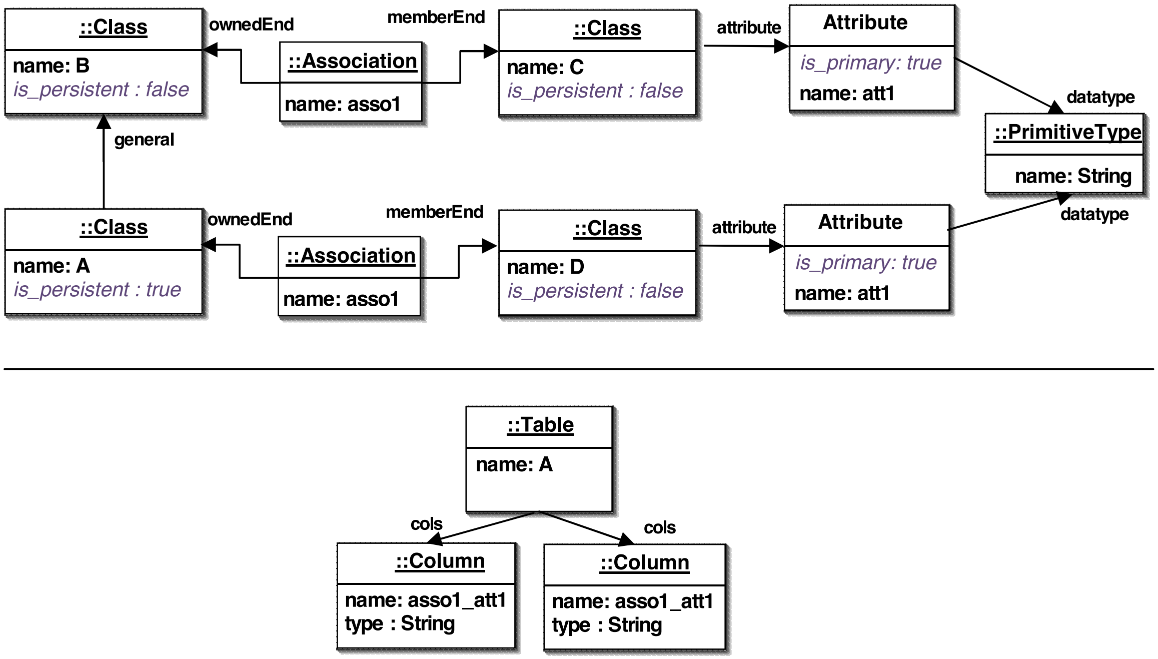


Figure 4.4: Model Excerpt for Pre-condition Improvement

may not have been well defined. In the following Section 4.1.3, we show how automatically generated models resulted in discovery of patterns that were not foreseen by experts who original designed the transformation class2rdbms.

After discovering pre-conditions that no longer lead to generation of models that are raise exceptions we regenerate sets of test models. We qualify the sets of generated test models via mutation analysis (see Section 4.1.4).

Pre-condition Improvement

The execution of a transformation helps us discover new constraints for the pre-condition $pre(MT)$ of the transformation MT. In this sub-section we illustrate how some of the constraints in the pre-condition of the transformation class2rdbms are discovered.

The discovery of a pre-condition starts with the detection of abnormal behaviour during the execution of automatically generated models. These may include exceptions such as memory leaks, infinite loops, or divide by zero errors. Models not previously considered by the model transformation specification often result in such exceptions. The exception handling mechanism in Kermeta allows us to detect and catch these exceptions. First, we prevent the lock of the execution when a transformation runs into infinite loop. For instance, this situation occurs when input models are navigated through a series of associations that can create loop structure in the transformation class2rdbms. These loops structures can navigate through diverse concepts such as inheritance trees, associations, and type of attributes. The Kermeta interpreter throws an

StackOverflowError exception when it detects such a problem.

Second, we detect other inconsistencies when output models produced from an automatically generated input model are not in the output domain. The output domain specified by an output metamodel MM_O, set of invariants on it C_O, and a post-condition $post(MT)$. In our case study, the transformation class2rdbms can produce ill-formed RDBMS models outside the valid output domain. A typical example is when a table contains several columns with same name. We detect these inconsistencies by checking if output models conform to the output metamodel (Ecore model of the metamodel with invariants) and satisfy post-conditions of the model transformation. The Figure 4.4 illustrates this detection. It represents an excerpt (bottom part) of an output model produced by the original transformation of a generated model (excerpt on the top part).

Our tool isolates inconsistent output models and corresponding input models. We then use a traceability mechanism and tool such as in [60] to restrain the analysis of these models on excerpts such as the one illustrated in Figure 4.4. Class named *A* is transformed into one table because it is persistent. It redefined an association of the Class *B*. Two associations with the same name *asso1* point to classes with the same attribute/property *att1*. Respecting the specification, the original transformations produces a table with two columns named *asso1_att1*. This does not conform to the RDBMS metamodel and it is detected by our tool. Construction of such models can be prevented by generating objects with different names. We solve this inconsistency by creating a new pre-condition constraint that protects the transformation from executing such models. We also regenerate new models that satisfy the new pre-condition constraints. For instance, the faulty model excerpt in Figure 4.4 can help us produce a new pre-condition that states:

In the classes of an inheritance tree, two associations with the same name can't point to classes that have (or their parent) attributes with same names.

Several new pre-conditions were discovered for the class2rdbms case study. We enlist nine newly discovered ALLOY facts in Appendix 6.6 apart from the initial set of pre-condition constraints as shown in Appendix 6.5. These ALLOY facts can be easily expressed in OCL to improve the pre-condition specification of class2rdbms. The conditions may even be applicable to commercial implementations of class2rdbms.

4.1.4 Qualifying Models: Mutation Analysis for Model Transformation Testing

We generate sets of test models using different strategies and qualify these sets via mutation analysis [49]. Mutation analysis involves creating a set of faulty versions or *mutants* of a program. A test set must distinguish the program output from all the output of its mutants. In practice, faults are modelled as a set of mutation operators where each operator represents a class of faults. A mutation operator is applied to the program under test to create each mutant. A mutant is killed when at least one test model detects the pre-injected fault. It is detected when program output and mutant output are different. A test set is relatively adequate if it kills all mutants of the original program. A mutation score is associated to the test set to measure its effectiveness in terms of percentage of the killed/revealed mutants.

We use the mutation analysis operators for model transformations presented in our previous work [107]. These mutation operators are based on three abstract operations linked to the basic

Table 4.1: Repartition of the class2rdbms mutants depending on the mutation operator applied

Mutation Operator	CFCA	CFCD	CFCP	CACD	CACA	RSMA	RSMD	ROCC	RSCC	Total
Number of Mutants	19	18	38	11	9	72	12	12	9	200

treatments in a model transformation: the navigation of the models through the relations between the classes, the filtering of collections of objects, the creation and the modification of the elements of the output model. Using this basis we define several mutation operators that inject faults in model transformations:

Relation to the same class change (RSCC): The navigation of one association toward a class is replaced with the navigation of another association to the same class.

Relation to another class change (ROCC): The navigation of an association toward a class is replaced with the navigation of another association to another class.

Relation sequence modification with deletion (RSMD): This operator removes the last step off from a navigation which successively navigates several relations.

Relation sequence modification with addition (RSMA): This operator does the opposite of RSMD, adding the navigation of a relation to an existing navigation.

Collection filtering change with perturbation (CFCP): The filtering criterion, which could be on a property or the type of the classes filtered, is disturbed.

Collection filtering change with deletion (CFCD): This operator deletes a filter on a collection; the mutant operation returns the collection it was supposed to filter.

Collection filtering change with addition (CFCA): This operator does the opposite of CFCD. It uses a collection and processes an additional filtering on it.

Class compatible creation replacement (CCCR): The creation of an object is replaced by the creation of an instance of another class of the same inheritance tree.

Classes association creation deletion (CACD): This operator deletes the creation of an association between two instances.

Classes association creation addition (CACA): This operator adds a useless creation of a relation between two instances.

Using these operators, we produced two hundred mutants from the class2rdbms model transformation with the repartition indicated in Table 4.1.

In general, not all mutants injected become faults as some of them are equivalent and can never be detected. The controlled experiments presented in this empirical study uses mutants presented in our previous work [107]. We have clearly identified faults and equivalent mutants to study the effect of our generated test models.

4.1.5 Test Strategies

Good strategies to guide automatic model generation are required to obtain test models that detect bugs in a model transformation. We define a strategy as a process that generates ALLOY *predicates* which are constraints added to the ALLOY model synthesized by CARTIER as described in Section 4.1.3. This combined ALLOY model is solved and the solutions are transformed to model instances of the input meta-model that satisfy the predicate. We present the following strategies to guide model generation:

- **Random/Unguided Strategy:** The basic form of model generation is unguided where only the Alloy model obtained from the meta-model and transformation is used to generate models. No extra knowledge is supplied to the solver in order to generate models. The strategy yields an empty Alloy predicate as shown in Listing 4.1.

```
pred random { }
```

Listing 4.1: Empty Alloy Predicate

- **Input-domain Partition based Strategies:** We guide generation of models using test criteria to combine *partitions* on domains of all properties of a meta-model (cardinality of references or domain of primitive types for attributes). A *partition* of a set of elements is a collection of *n* ranges A_1,..., A_n such that A_1, ..., A_n do not overlap and the union of all subsets forms the initial set. These subsets are called *ranges*. We use partitions of the input domain since the number of models in the domain are infinitely many. Using partitions of the properties of a meta-model we define two test criteria that are based on different strategies for combining partitions of properties. Each criterion defines a set of *model fragments* to cover an input meta-model. These fragments are transformed to predicates on meta-model properties by Cartier. For a set of test models to cover the input domain at least one model in the set must cover each of these model fragments. We generate model fragment predicates using the following test criteria to combine partitions (cartesian product of partitions):
  - **AllRanges Criteria:** AllRanges specifies that each range in the partition of each property must be covered by at least one test model.
  - **AllPartitions Criteria:** AllPartitions specifies that the whole partition of each property must be covered by at least one test model.

The notion of test criteria to generate model fragments was initially proposed in the paper [55]. The accompanying tool called Meta-model Coverage Checker (MMCC) [55] generates model fragments using different test criteria taking any meta-model as input. Then, the tool automatically computes the coverage of a set of test models according to the generated model fragments. If some fragments are not covered, the set of test models should be improved in order to reach a better coverage.

In this study, we use the model fragments generated by MMCC for the UMLCD Ecore model (Figure 4.2). We use the criteria AllRanges and AllPartitions. For example, in Table 4.2, *mfAllRanges1* and *mfAllRanges2* are model fragments generated by Cartier using MMCC [55] for the *name* property of a classifier object. The *mfAllRanges1* states that there must be at least one classifier object with an empty name while *mfAllRanges2* states that there must be at least one classifier object with a non-empty name. These values for name are the ranges for the property. The model fragments chosen using AllRanges *mfAllRanges1* and *mfAllRanges2* define two partitions *partition1* and *partition2*. The model fragment *mfAllPartitions1* chosen using AllPartitions defines both *partition1* and *partition2*.

These model fragments are transformed to Alloy predicates by Cartier. For instance, model fragment *mfAllRanges7* is transformed to the predicate in Listing 4.2.

```
pred mfAllRanges7
{
  some c : Class | #c.attribute=1
}
```

Listing 4.2: ALLOY Predicate for *mfAllRanges7*

As mentioned in our previous work [55] if a test set contains models where all model fragments are contained in at least one model then we say that the input domain is completely covered. However, these model fragments are generated considering only the concepts and relationships in the Ecore model and they do not take into account the constraints on the Ecore model. Therefore, not all model fragments are consistent with the input meta-model because the generated models that contain these model fragments do not satisfy the constraints on the meta-model. CARTIER invokes the ALLOY Analyzer [72] to automatically check if a model containing a model fragment and satisfying the input domain can be synthesized for a general scope of number of objects. This allows us to *detect inconsistent model fragments*. For example, the following predicate, *mfAllRanges7a*, is the ALLOY representation of a model fragment specifying that some Class object does not have any Property object. CARTIER calls the ALLOY API to execute the run statement for the predicate *mfAllRanges7a* along with the base ALLOY model to create a model that contains up to 30 objects per class/concept/signature (see Listing 4.3).

Table 4.2: Consistent Model Fragments Generated using AllRanges and AllPartitions Strategies

Model-Fragment	**Description**
mfAllRanges1	A Classifier $c \mid c.name =$“”
mfAllRanges2	A Classifier $c \mid c.name! =$“”
mfAllRanges3	A Class $c \mid c.is\_persistent = True$
mfAllRanges4	A Class $c \mid c.is\_persistent = False$
mfAllRanges5	A Class $c \mid \#c.general = 0$
mfAllRanges6	A Class $c \mid \#c.general = 1$
mfAllRanges7	A Class $c \mid \#c.attribute = 1$
mfAllRanges8	A Class $c \mid \#c.attribute > 1$
mfAllRanges9	An Property $a \mid a.is\_primary = True$
mfAllRanges10	An Property $a \mid a.name =$“”
mfAllRanges11	An Property $a \mid a.name! =$“”
mfAllRanges12	An Property $a \mid \#a.datatype = 1$
mfAllRanges13	An Association $as \mid as.name =$“”
mfAllRanges14	An Association $as \mid \#as.memberEnd = 0$
mfAllRanges15	An Association $as \mid \#as.memberEnd = 1$
mfAllPartitions1	Classifiers $c1, c2 \mid c1.name =$“” and $c2.name! =$“”
mfAllPartitions2	Classes $c1, c2 \mid c1.is\_persistent = True$ and $c2.is\_persistent = False$
mfAllPartitions3	Classes $c1, c2 \mid \#c1.general = 0$ and $\#c2.general = 1$
mfAllPartitions4	Propertys $a1, a2 \mid a1.is\_primary = True$ and $a2.is\_primary = False$
mfAllPartitions5	Associations $as1, as2 \mid as1.name =$“” and $as2.name! =$“”

```
pred mfAllRange7a
{
  some c:Class | #c.attribute = 0
}

run mfAllRanges7 for 30
```

Listing 4.3: ALLOY Predicate and Run Command

The ALLOY analyzer yields a *no solution* to the run statement indicating that the model fragment is not consistent with the input domain specification. This is because no model can be created with this model fragment that also satisfies an input domain constraint that states that every Class must have at least one Property object as shown in Listing 4.4.

```
sig Class extends Classifier
{ ...
  attribute : some Property
  ...
}
```

Listing 4.4: Example ALLOY Signature

In Listing 4.4, *some* indicates 1..*. However, if a model solution can be found using ALLOY we call it a *consistent model fragment*. MMCC generates a total of 15 consistent model fragments using AllRanges and 5 model fragments using the AllPartitions strategy, as shown in Table 4.2.

4.1.6 Experiments

Experimental Setup and Execution

We use the methodology in Section 4.1.3 to compare coverage based test generation with unguided/random test model generation.

We generate sets of test models based on factorial experimental design [121]. We consider the *exact number of objects for each class* in the effective input meta-model as factors for experimental design. A factor level is the exact number of objects of a given class in a test model. These factors help study the effect of number of different types of objects on the mutation score. For instance, we can ask questions such as whether a large number of Association objects have a correlation with the mutation score? The large number of Association objects also indicates a highly connected UML class diagram test model. We decide these factor levels by simple experimentation such as verifying if models can be generated in reasonable amount of time given that we need to generate thousands of test models in a few hours. We also want to cover a combination of a large number of varying factor levels. We have 8 different factor levels for the different classes in the UML class diagram effective input meta-model as shown in Table 4.3. Other factors that may affect but are not considered for test model generation are the use different SAT solvers such as SAT4J, MiniSAT, or ZChaff, maximum time to solve, t-wise interaction between model fragments.

The AllRanges criteria on the UMLCD meta-model gives 15 consistent model fragments (see Table 4.2). We have 150 models in a set, where 10 non-isomorphic models satisfies each different model fragment. We generate 10 non-isomorphic models to verify that mutation scores do

Table 4.3: Factors and their Levels for Test Sets

Factors	S1	S2	S3	S4	S5	S6	S7	S8
#ClassModel	1	1	1	1	1	1	1	1
#Class	5	5	15	15	5	15	5	15
#Association	5	15	5	15	5	5	15	15
#Attribute	25	25	25	25	30	30	30	30
#PrimitiveDataType	4	4	4	4	4	4	4	4
Bit-width Integer	5	5	5	5	5	5	5	5
#Models/Set AllRanges	15	15	15	15	15	15	15	15
#Models/Set Unguided	15	15	15	15	15	15	15	15
#Models/Set AllPartitions	5	5	5	5	5	5	5	
#Models/Set Unguided	5	5	5	5	5	5	5	

Table 4.4: Mutation Scores in Percentage for All Test Model Sets

Set	1	2	3	4	5	6	7	8
Unguided **150 models/set in 8 sets**	68.56	69.9	68.04	70.1	70.1	68.55	69	70.1
AllRanges **150 models/set in 8 sets**	88.14	92.26	81.44	85	91.23	80.4	91.23	88.14
Unguided **50 models/set in 8 sets**	70.1	62.17	68.04	70.1	65.46	68.04	69.94	70.1
AllPartitions **50 models/set in 8 sets**	90.72	93.3	84.53	87.62	87.62	82.98	92.78	88.66

not drastically change within each solution. We synthesize 8 sets of 150 models using different levels for factors as shown in Table 4.3 (see rows 1,2,3,4,5,6). The total number of models in these 8 sets is 1200.

The AllPartitions criteria gives 5 consistent model fragments. We have 50 test models in a set, where 10 non-isomorphic test models satisfies a different model fragment. We synthesize 8 sets of 50 models using factor levels shown in Table 4.3. The levels for factors for AllRanges and AllPartitions are the same. Total number of models in the 8 sets is 400. The selection of these factors at the moment is not based on a problem-independent strategy.

We compare test sets generated using AllRanges and AllPartitions with unguided test sets. For each test set of coverage based strategies we generate an equal number of random/unguided models as a reference to qualify the efficiency of different strategies. Precisely, we have 8 sets of 150 unguided test models to compare with AllRanges and 8 sets of 50 unguided test models to compare with AllPartitions. We use the factor levels in Table 4.3.

To summarize, we generate a total of 3200 models using an Intel(R) Core^{TM} 2 Duo processor with 4GB of RAM. We perform mutation analysis of these sets to obtain mutation scores on a grid of 10 Intel Celeron 440 high-end computers. The computation time for generating 3200 models was about 3 hours and mutation analysis took about 1 week. We discuss the results of mutation analysis in the following section.

Results and Discussion

Mutation scores for AllRanges test sets are shown in Table 4.4 (row 2). Mutation scores for test sets obtained using AllPartitions are shown in Table 4.4 (row 4). We discuss the effects of the influencing factors on the mutation score:

- The number of Class objects and Association objects are factors that have a strong correlation with the mutation score. This is due to a specific characteristic of the transformation. The transformation class2rdbms principally transforms all persistent classes in an UML model to tables in RDBMS and all attributes/associations to columns. Therefore, the probability of finding a fault that process classes and associations is high. We notice this correlation due to an increase in mutation score with the level of these factors. This is true for sets from unguided and model fragments based strategies. For instance, the lowest mutation score using AllRanges is 80.41 %. This corresponds to set 1 where the factor levels are 1,5,5,25,4,5 (see Column for set 1 in Table 4.3) and highest mutation scores are 91,24 and 92,27% where the factor levels are 1,15,5,25,4,5 and 1,5,15,25,4,5 respectively (see Columns for set 3 and set 7 in Table 4.3).
- We observe that AllPartitions test sets containing only 50 models/set gives a score of maximum 93.3%. The AllPartitions strategy demonstrates that knowledge from two different partitions satisfied by one test model greatly improves bug detecting efficiency. This also opens a new research direction to explore: Finding strategies to combine model fragments to guide generation of smaller sets of complex test models with better bug detecting effectiveness.

We compare unguided test sets with model fragment guided sets in the *box-whisker* diagram shown in Figure 4.5. The box whisker diagram is useful to visualize groups of numerical data such as mutation scores for test sets. Each box in the diagram is divided into lower quartile (25%), median, upper quartile (75% and above), and largest observation and contains statistically significant values. A box may also indicate which observations, if any, might be considered outliers or whiskers. In the box whisker diagram of Figure 4.5 we shown 4 boxes with whiskers for unguided sets and sets for AllRanges and AllPartitions. The X-axis of this plot represents the strategy used to select sets of test models and the Y-axis represents the mutation score for the sets.

We make the following observations from the box-whisker diagram:

- Both the boxes of AllRanges and AllPartitions represent mutation scores higher than corresponding unguided sets.
- The high median mutation scores for strategies AllRanges 88.14% and AllPartitions 88.14% indicate that both these strategies return consistently good test sets.
- The small size of the box for AllPartitions compared to the AllRanges box indicates its relative convergence to good sets of test models.
- The small set of 50 models using AllPartitions gives mutations scores equal or greater than 150 models/set using AllRanges. This implies that it is a more efficient strategy for test

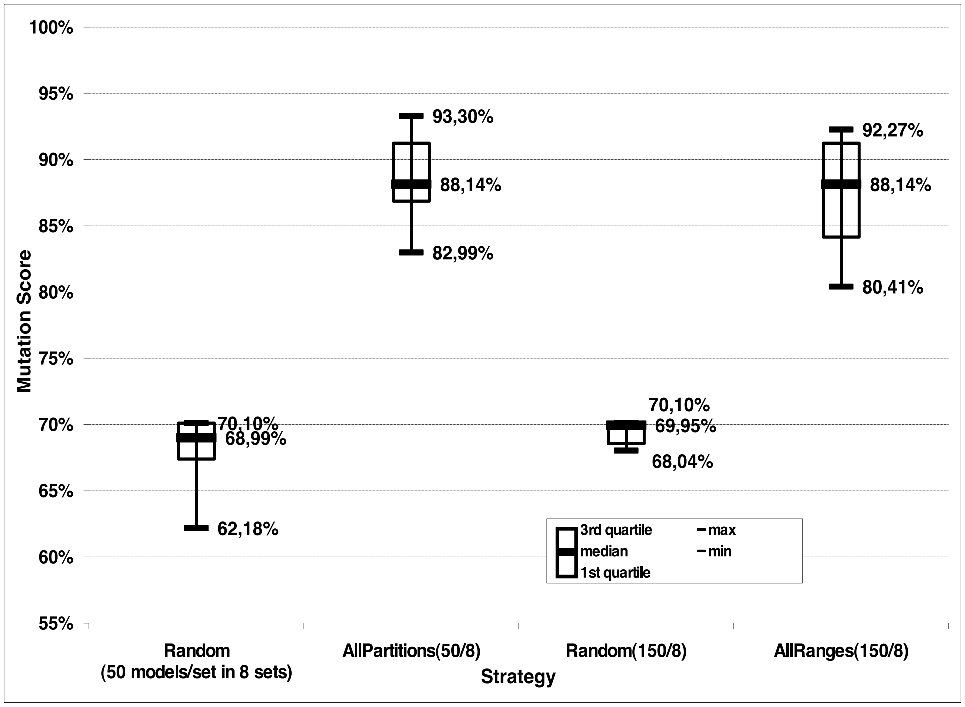


Figure 4.5: Box-whisker Diagram to Compare Automatic Model Generation Strategies

model selection. The main consequence is a reduced effort to write corresponding *test oracles* [107] with 50 models compared to 150 models.

- Despite the generation of multiple solutions (10 solutions for each model fragment or an empty fragment for unguided generation) for each strategy we see a consistent behaviour in the mutation scores. There is no large difference in the mutation scores especially for unguided generation. The median is 69% and the mutation scores range between 68% and 70%. The AllRanges and AllPartitions vary a little more in their mutation scores due to a larger coverage of the effective input meta-model.

The freely and automatically obtained knowledge from the input meta-model using the MMCC algorithm shows that AllRanges and AllPartitions are successful strategies to guide test generation. They have higher mutation scores with the same sources of knowledge used to generate unguided test sets. A manual analysis of the test models reveals that injection of inheritance via the parent relation in model fragments results in higher mutation scores. Most unguided models do not contain inheritance relationships as it is not imposed by the meta-model.

What about the 7% of the mutants that remain alive given that the highest mutation score is 93.3%? We note by an analysis of the live mutants that they are the same for both AllRanges and AllPartitions. There remain 19 live mutants in a total of 200 injected mutants (with 6 equivalent mutants). In the median case both AllRanges and AllPartitions strategy give a mutation score of 88.14%. The live mutants in the median case are mutants not killed due to fewer objects in models.

To consistently achieve a higher mutation score we need more CPU speed, memory and parallelization to efficiently generate larger test models and perform mutation analysis on them. This extension of our work has not be been explored by us. It is important for us to remark that some live mutants can only be killed with more information about the model transformation such as those derived from its requirements specification. For instance, one of the remaining live mutant requires a test model with a class containing several primitive type attributes such that at least one is a primary attribute. A test model that satisfies such a requirement requires the combination of model fragments imposing the need for several attributes in a class A, attributes of class A must have primitive types, at least one primary attribute in the class A, and at least one non-primary attribute in the class A. This requirement can either be specified by manually creating a combination of fragments or by developing a better general test strategy to combine multiple model fragments. In another situation, we observe that not all model fragments are consistent with the input domain and hence they do not really cover the entire meta-model. Therefore, we miss killing some mutants. This information could help improve partitioning and combination strategies to generate better test sets.

4.1.7 Conclusion for Test Generation

Black-box testing exhibits the challenging problem of developing efficient model generation strategies. In this empirical study we use CARTIER to generate models conforming to the input domain and guided by different test strategies. First, CARTIER helps us precisely specify the input domain of a model transformation via meta-model pruning and pre-condition improvement.

Second, we use CARTIER to generate sets of test models that compare coverage and unguided strategies for model generation. All test sets using these strategies detect faults given by their mutation scores. The comparison of coverage strategies with unguided generation taught us that both strategies AllPartitions and AllRanges look very promising. Coverage strategies give a maximum mutation score of 93% compared to a maximum mutation score of 70% in the case of unguided test sets. We observe that mutation scores do not vary drastically despite the generation of multiple solutions for the same test strategy. We conclude from our experiments that the AllPartitions strategy is a promising strategy to consistently generate a small test of test models with a good mutation score. However, to improve efficiency of test sets we might require effort from the test designer to obtain test model knowledge/test strategy that take the internal model transformation design requirements into account.

4.2 Towards Model Completion in Domain-specific Model Editors

Documents in the form of computer programs, diagrams, chemical formulas, and markup text can currently be edited in document editors called *structure editors*. These structure editors are cognizant of the document's underlying structure such as the grammatical syntax or a formal grammar of the language. Functionally, these structure editors are syntax or language-directed to aid the user by presenting recommendations for completion of code, text, or a diagram based on correct possibilities prescribed by the underlying structure. This enables faster document development with fewer errors. However, structure editors are separately constructed for each domain-specific language and are built mainly for grammar-based textual languages. We are interested in the subject of extending structure editors from high-level models built using the principles of *Model Driven Engineering* (MDE) [57] where domain-specific model editors are automatically synthesized for a variety of modelling languages.

In MDE, given a meta-model specification of a domain-specific modelling language, software tools can automatically generate *domain-specific model editors*. For example, generative modelling tools such as AToM^3 (A Tool for Multi-formalism Meta-modelling) [48][67],GME (Generic Modelling Environment)[12], GMF (Eclipse Graphical Modelling Framework)[79] can synthesize a domain-specific visual model editor from a declarative specification of a domain-specific modelling language. A declarative specification consists of a meta-model and a visual/textual syntax that describes how language elements (objects and relationships) manifest in the model editor. The designer of a model uses this model editor to construct a model on a drawing canvas. This is analogous to using an integrated development environment (IDE) to enter a program or a word processor to enter sentences. However, IDEs such as Eclipse present recommendations for completing a program statement when possible based on its grammar and existing libraries [15]. Similarly, Microsoft Word presents grammatical correction recommendations if a sentence does not conform to a natural language grammar. Therefore, we ask: Can we extrapolate similar technology or develop new technology for partial models constructed in a model editor for a domain-specific modelling language (DSML)?

Extrapolating code completion techniques for model completion is not feasible in the general case. The first reason is the difference between the underlying structure of code and models. Code completion techniques use the Backus-Naur Form (BNF) grammar of a programming language while models are specified by a meta-model and constraints on it. Second, model completion must consider completing the entire model as constraints can span entire models unlike code completion which presents solutions at a program statement level. Third, in terms of reduction in effort model completion must help reduce the effort of a modeller by automatically satisfying all relevant language constraints since in general they may be too hard for a modeller to resolve manually. The output of model completion must be one or many valid models that conform to their language. This notion of reduction in effort is different from that in code completion. Code completion presents local suggestions to complete navigational expressions or concept names but it does not perform constraint satisfaction to output a valid program. In the general case, model completion may take more time than code or sentence completion which are almost instantaneous. Therefore, there is a need to develop new techniques for model completion with different goals such as relaxing the exigence towards time to complete.

The major difficulty for providing completion capabilities in model editors is to integrate heterogeneous sources of knowledge in the computation of the possible solutions for completion. The completion algorithm must take into account the concepts defined in the meta-model, constraints on the concepts and the partial model built by a domain expert/user. The difficulty is that these three sources of knowledge are obviously related (they refer to the same concepts) but are expressed in different languages, sometimes in different files, and in most cases by different people and at different moments in the development cycle as they are separable concerns.

In this section, we propose present a transformation from a partial model to an ALLOY[71] [72] predicate. The generated ALLOY predicate is included in the ALLOY model generated from the metamodel of a DSML. The transformation of a metamodel has been discussed in Chapter 3. The predicate is solved to obtain recommendations for completing the partial model in a model editor. Our transformation from the heterogeneous sources to ALLOY is integrated in the software tool AToM^3.

The *scientific contribution* in this section addresses two important questions:

- **Question 1:** How can we generate a complete model(s) from a partial model specification?
- **Question 2:** How can we integrate a model completion mechanism in a domain-specific model editor?

The precise contributions of this section addresses exactly these problems. We enlist them below:

- **Contribution 1:** First, the DSML metamodel and its invariants in transformed to a base ALLOY [71] model using techniques already described in Chapter 3. In this section present a transformation from a partial model to an ALLOY predicate and concatenate it to the base ALLOY model. The predicate representing the partial model is solved in the resulting ALLOY model to generate complete models that conform to the metamodel specification.
- **Contribution 2:** We integrate this model completion mechanism into the metamodeling environment AToM^3 such that any DSML generated using AToM^3 by construction comes with model completion. Users can create partial models in a DSML generated using AToM^3

and automatically obtain recommendations to complete them by clicking on a button. The complete models are shown in the concrete visual syntax of the DSML.

An overview of our methodology is presented in Section 4.2.1. One of the key parts of our methodology is the automatic synthesis of domain-specific model editors from their specification comprising of the meta-model and visual syntax. This process is described in Section 4.2.2. The component that will add model completion ability to the synthesized model editor is a transformation from a partial model to an ALLOY predicate. We present this transformation in Section 4.2.6. Once we include this transformation into the synthesis of a domain-specific model editor we are able to synthesize domain-specific model editors with automatic model completion. We describe the model completion process in Section 4.2.8. We present examples of model completion recommendations generated for partial models in Section 4.2.9. We conclude in Section 4.2.10.

4.2.1 Methodology for Model Completion

The development and use of a domain-specific model editor with automatic model completion can be divided into the following phases and sub-phases:

1. Specification of a domain-specific modelling language (in AToM^3)
    (a) Specification of a metamodel
        i. Specification of a class diagram (Ecore model)
        ii. Specification of facts on the concepts in the class diagram (ALLOY facts in our case)
    (b) Specification of a visual syntax in an icon editor (available in AToM^3) for concepts in the metamodel
2. Transformation of metamodel and visual syntax to a model editor
    (a) Synthesis of an editor with buttons, menus and icons
    (b) Synthesis of a drawing canvas with features such as automatic layout
    (c) Synthesis of a clickable widget for model completion
    (d) Synthesis of a dialog box for specifying model completion parameters
3. User interaction
    (a) Drawing a partial model on the canvas
    (b) Editing model completion parameters
    (c) Click on a button to generate complete model(s)
4. Model Completion (hidden from user)
    (a) Transformation to a base ALLOY model from the Ecore model

(b) Augmenting metamodel facts with base ALLOY model

(c) Synthesis of an ALLOY predicate from partial model and augmentation to base ALLOY model

(d) Synthesis of run commands from the model completion parameters and augmentation to current ALLOY model

(e) Solving final ALLOY model and returning complete models as recommendations to the model editor

The specification of a domain-specific language is usually done by a *language designer* who interacts with domain experts to identify the concepts, their properties and relationships in a domain of knowledge, science or engineering. The language designer also develops a repository of constraints among the concepts and its properties. The assembly of the concepts and relationships is expressed as an Ecore model by the language designer. The constraints on the Ecore model or class diagram (CD) are expressed in a formal constraint language. Preferably, a constraint language that has a finite number of solutions and is decidable. In our methodology we use *facts* expressed in the language ALLOY to represent such constraints. The CD and the set of constraints on it results in the *metamodel* of a Domain-specific Modelling Language (DSML)

A *visual syntax designer* specifies a concrete visual syntax for the concepts and relationships in the modelling language. In our methodology we use the AToM^3 icon editor to specify a visual syntax. In Section 4.2.2 we discuss in detail the specification of the modelling language for Finite State Machines (FSM) along with a visual syntax.

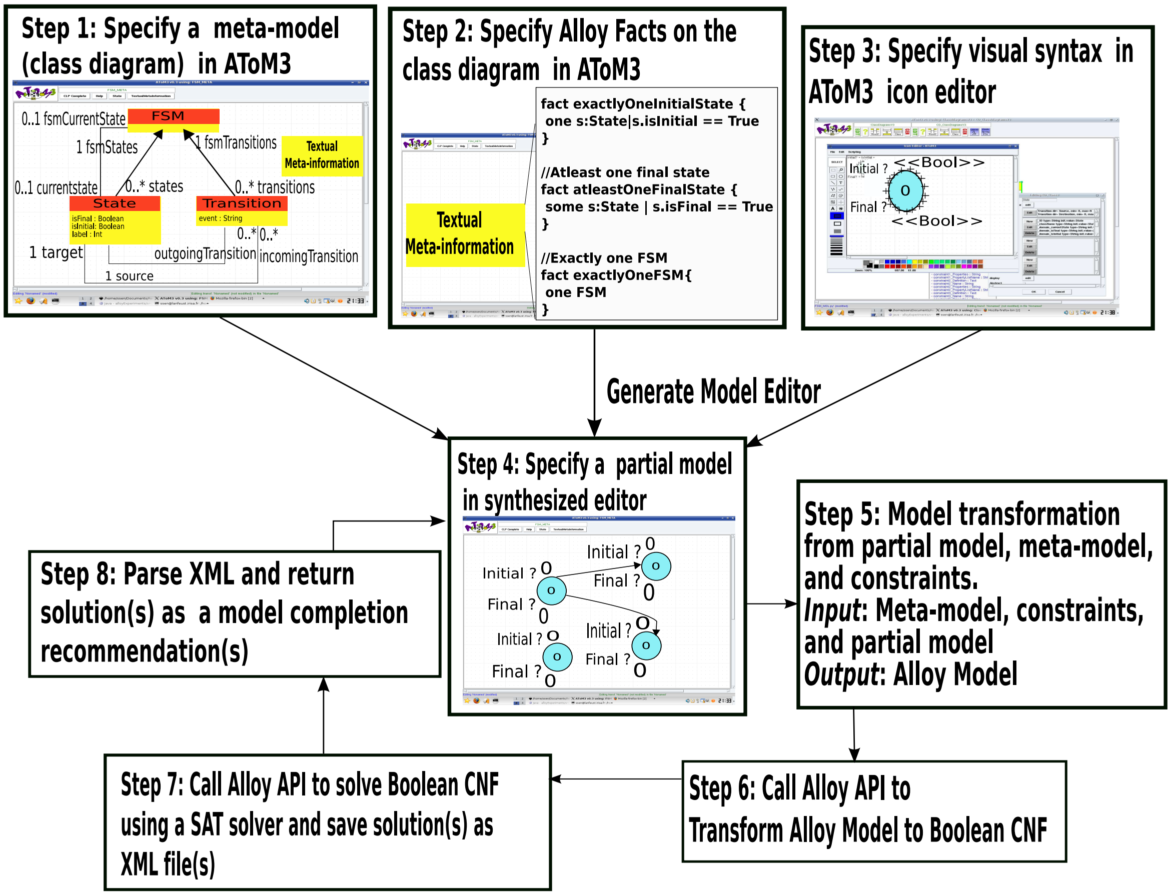
Once we have all the elements (metamodel and visual syntax) necessary for a domain-specific modelling language a *model transformation engineer* develops a transformation to synthesize a visual domain-specific model editor from these elements. The model editor consists of buttons, menus, and a canvas. A user can select and drop objects on a drawing canvas and connect them using relationships. The objects are manifested as icons as specified in the icon editor for the concrete visual syntax by the visual syntax designer. The relationships are links between these icons. In the model editor by clicking on the icon the user can edit or specify the values of properties.

In our work, we extend this model transformation by transforming the metamodel to an ALLOY model (see Chapter 3). The transformation also synthesizes a button widget in the domain-specific model editor. A *domain expert* or *user* can click on this button resulting in the solving of the ALLOY model augmented with ALLOY predicates synthesized from the partial model drawn on the canvas. Recommendations as one or more complete models (if found) are returned to the model drawing canvas. In Section 4.2.6 we present the transformation from a partial model to ALLOY. An illustrative outline of the model completion methodology is shown in Figure 4.6.

4.2.2 Specifying a DSML

4.2.3 Metamodel

The first step in specifying a DSML is creating a metamodel for a modelling language. The metamodel for the FSM modelling language is presented in Figure 4.7. The classes in the metamodel


Step 1: Specify a meta-model (class diagram) in AToM3
0..1 fsmCurrentState
FSM
1 fsmStates
1 fsmTransitions
Textual Meta-information
0..1 currentstate
0..* states
0..* transitions
State
isFinal : Boolean
isInitial: Boolean
label : Int
Transition
event : String
0..* 0..*
1 target
outgoingTransition
incomingTransition
1 source
Step 2: Specify Alloy Facts on the class diagram in AToM3
Textual Meta-information
Step 3: Specify visual syntax in AToM3 icon editor
Initial ?
<<Bool>>
Final ?
<<Bool>>
Generate Model Editor
Step 4: Specify a partial model in synthesized editor
Initial ?
Final ?
Step 5: Model transformation from partial model, meta-model, and constraints.
Input: Meta-model, constraints, and partial model
Output: Alloy Model
Step 6: Call Alloy API to Transform Alloy Model to Boolean CNF
Step 7: Call Alloy API to solve Boolean CNF using a SAT solver and save solution(s) as XML file(s)
Step 8: Parse XML and return solution(s) as a model completion recommendation(s)

```
fact exactlyOneInitialState {
 one s:State|s.isInitial == True
}

//Atleast one final state
fact atleastOneFinalState {
 some s:State | s.isFinal == True
}

//Exactly one FSM
fact exactlyOneFSM{
 one FSM
}
```

Figure 4.6: Methodology Overview

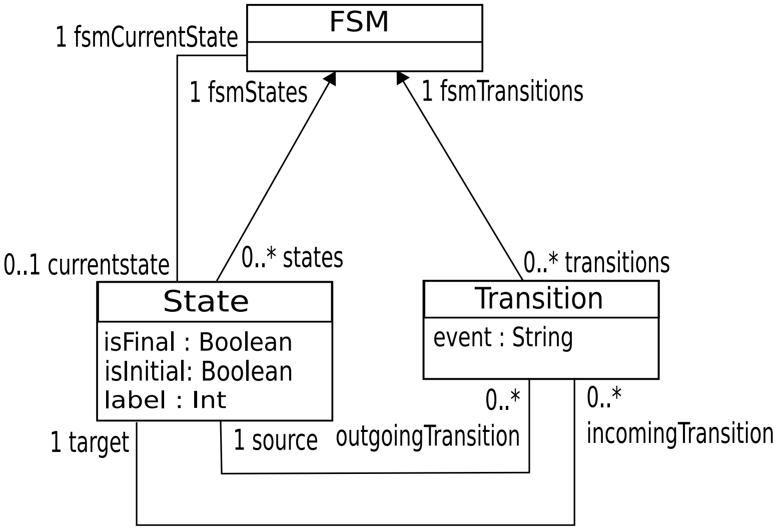


Figure 4.7: The Finite State Machine Metamodel

are FSM, State and Transition. The metamodel is specified using the Ecore industry standard.

4.2.4 Constraints on Metamodel

The second step comprises of specifying constraints on the metamodel. We directly specify ALLOY facts on the FSM metamodel. These ALLOY facts were manually transformed from original OCL constraints on FSM. In Table 4.5, we present the constraints on the FSM metamodel in natural language and as ALLOY facts.

In the appendix we present the complete ALLOY model for the FSM modelling language. This ALLOY model can be loaded into the ALLOY Analyzer [72] for directly obtaining valid FSM models.

4.2.5 Visual Syntax

The final step (in specifying a DSML for synthesizing a model editor) we take is to specify the concrete visual syntax of the class of objects in the metamodel. The visual syntax specifies what an object looks like on a 2D canvas. An icon editor in AToM^3 is used to specify the visual syntax of the classes in the metamodel.

An icon editor is used to specify the visual syntax of metamodel concepts such as classes and relationships. The icon for State is a circle annotated with three of its attributes (isFinal, isInitial, and label). The connectors in the diagram are points of connection between State objects and Transition objects.

The visual syntax can also by dynamically changed based on the properties of the model. In an iconic visual modelling language such as FSM, the first step taken in specifying a visual syntax is drawing an icon that represents a class of objects. If needed it is annotated with text

Table 4.5: Constraints in natural language and as ALLOY facts

<table>
<tr><th>Constraint Name and Definition</th><th>Alloy Fact</th></tr>
<tr><td><b>exactlyOneFSM</b>: There must be exactly one FSM object in a FSM model</td><td><pre>fact exactlyOneFSM
{
 one FSM
}</pre></td></tr>
<tr><td><b>atleastOneFinalState</b> : There must be at least one final state in a FSM model</td><td><pre>fact at leastOneFinalState
{
 some s:State|s.isFinal==True
}</pre></td></tr>
<tr><td><b>exactlyOneInitialState</b> : There must be exactly one initial state in the FSM model</td><td><pre>fact exactlyOneInitialState
{
 one s:State|s.isInitial==True
}</pre></td></tr>
<tr><td><b>sameSourceDiffTarget</b> : All transitions with the same source must have different target</td><td><pre>fact sameSourceDiffTarget
{
  all t1:Transition,t2:Transition|
  (t1!=t2 and t1.source==t2.source) =>
  t1.target!=t2.target
}</pre></td></tr>
<tr><td><b>setTargetAndSource</b> : The target of an incoming transition to a State itself and the source of all its outgoing transitions is the same State</td><td><pre>fact setTargetAndSource
{
 all s:State |
 s.incomingTransition.target = s and
 s.outgoingTransition.source=s
}</pre></td></tr>
<tr><td><b>noUnreachableStates</b>: There can be no unreachable states in the FSM from an initial state. Since, its a ternary constraint we approximate it by stating that a non-initial state can be reached from an initial state up to a maximum depth of N (N=3 is the given example).</td><td><pre>fact noUnreachableStates
{
 all s:State| (s.isInitial==False) =>
 #s.incomingTransition >=1 and
 (s.isInitial==True and #State > 1) =>
 #s.outgoingTransition >=1 and
 s.outgoingTransition.target!=s
}</pre></td></tr>
<tr><td><b>uniqueStateLabels</b> : All State objects have unique labels</td><td><pre>fact uniqueStateLabels
{
 #State>1 => all s1:State,s2:State |
 s1!=s2=>s1.label != s2.label</pre></td></tr>
</table>

and its properties. Connectors are added to the visual object so that it can be connected to other objects if they are related.

4.2.6 Transformation of a Partial Model

We define a partial model as a graph of objects such that: (1) The objects are instances of classes in the modelling language metamodel (2) The partial model either does not conform to the language metamodel or its invariants expressed in a textual constraint language. A complete model on the other hand contains all the objects of the partial model and additional objects or property value assignments in new/existing objects such that it conforms both to the metamodel and its invariants.

A partial model, such as in Figure 4.8 (a), is *automatically* transformed to a set of ALLOY predicates by navigating it object by object in the canvas. We navigate all objects of a certain type and put them together as an ALLOY predicate. We want to keep the already specified properties for each object in the partial model but also allow for extensibility. For instance, for all the State objects in the partial model of Figure 4.8 (a) we create an ALLOY predicate as shown in the first predicate of Figure 4.8 (b). The ALLOY predicate states that there exists at least one State object s1, at least one State object s2, at least one State object s3, at least one State object s4 (representing the four State objects in the partial model), at least one Transition object t1, and at least one Transition object t2 such that s1,s2,s3,s4 are not equal and t1,t2 are not equal. The predicate also states that the Transition objects t1 and t2 are in the set of outgoing transitions for State object s1. Transition object t1 is in the set of incoming transitions of s1. The Transition object t2 is in the set of incoming transitions of s2. These sets are open for inclusion of new Transition objects. These predicates preserve all knowledge coming from the partial model while allowing the extension to relations to more objects.

We present a procedure to describe the transformation from the partial model to a set of ALLOY predicates below:

The following represents the procedure to synthesize an ALLOY predicate from a partial model

1. We start by synthesizing the header of a partial model:

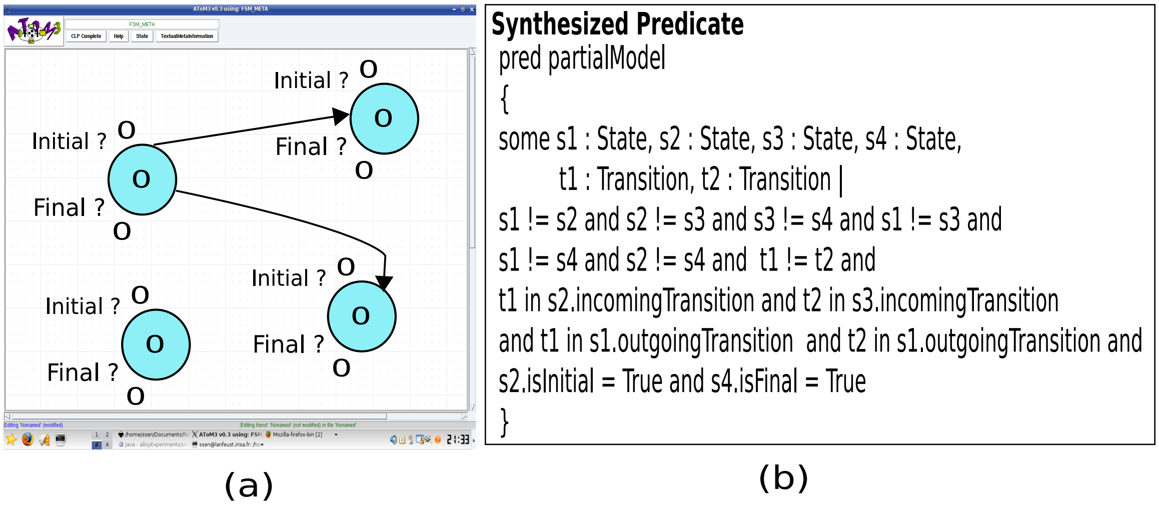
   ```
   pred  partialModel {
   ```

2. For all objects of o_{ij} of type $Class_j$ in a partial model we synthesize an ALLOY expression: some $o_{ij} : Class_j, ... \mid$

3. For all objects of o_{ij} of type $Class_j$ and all objects o_{kj} of type $Class_j$ in a partial model we synthesize an ALLOY expression:
   $o_{ij}! = o_{kj}$, each expression separated by *and*

4. For all defined attributes a_{ijk} of o_{ij} we synthesize the expression:
   $o_{ij}.a_{ijk} = v$, where v is the specified value separated by commas

5. For all defined references r_{ijk} of o_{ij} we synthesize the expression:
   v in $o_{ij}.r_{ijk}$, where v is the object in the set of referred objects separated by commas



Synthesized Predicate

```
pred partialModel
{
some s1 : State, s2 : State, s3 : State, s4 : State,
        t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and
s1 != s4 and s2 != s4 and  t1 != t2 and
t1 in s2.incomingTransition and t2 in s3.incomingTransition
and t1 in s1.outgoingTransition  and t2 in s1.outgoingTransition and
s2.isInitial = True and s4.isFinal = True
}
```

Figure 4.8: (a) Partial Model (b) Synthesized Predicates from Partial Model

6. We finish the predicate by closing the brace.

4.2.7 Transforming ALLOY Model Completion Parameters

The user is provided with a dialog box to insert *model completion parameters*. Model completion parameters include finite scopes such as the upper bound on the number of objects of any class, or the upper-bound on the number of objects for each class, or the exact number of objects for each class, or a mixture of upper bounds and exact number of objects for different classes. The default scope is number of objects in the partial model. An other parameter is the number of solutions required *S*. This information is used to synthesize an ALLOY *run command* that is finally inserted in the ALLOY model. For example, if the partial model predicate is called *partialModel1* and the user states that he wants exactly one object of class A, up to 10 objects of class B, and a scope of 5 for integers then the following run statements is synthesized:

```
run partialModel1 for exactly 1 A, 10 B, 5 Int
```

If the number of objects in the partial model is N, then the default run command the editor generates is:

```
run partialModel1 for N
```

4.2.8 Model Completion Process

The model completion process integrated in the domain-specific model editor takes as input the Ecore model, augmented ALLOY facts, and a partial model drawn in the model editor synthesized from the class diagram of a modelling language, and set of parameters to define the scope of the complete models to be synthesized. The process is invoked when a user draws a partial model in the modelling canvas and clicks on the *Generate Completion Recommendations* button.

The following steps are executed during the completion process :

1. An ALLOY model (ALS) file is synthesized containing the signature definitions of the classes in the Ecore model as described in Chapter 3.

2. The modelling language facts are augmented to the ALLOY model. These facts are specified as described in Section 4.2.4.

3. The partial model drawn in the model editor canvas is transformed to a predicate as described in Section 4.2.6 and augmented to the current ALLOY model

4. The model completion parameters are transformed to a run command (See Section 4.2.7) and augmented to the ALLOY model giving us an adequate description for model completion.

5. The model editor invokes a solver to generate complete model recommendations for the partial model.

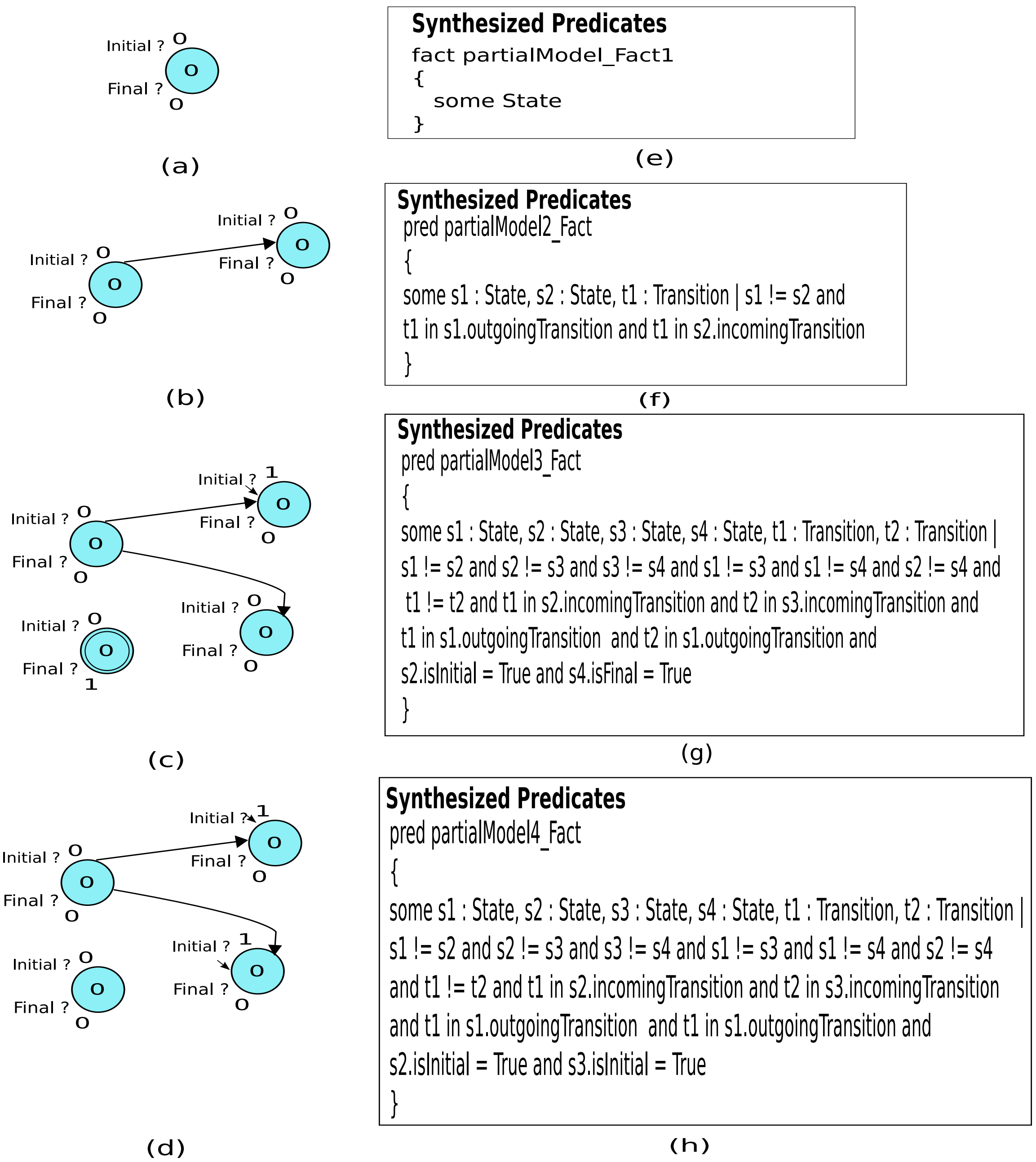
It is important to note that the partial model is specified as a source of knowledge about what objects and properties that the user wants to absolutely see in the complete model. In the complete model we can see the intact contents of the partial model. However, the object identifiers of the partial model are not preserved in the complete model. We also do not perform pattern matching to identify the original partial model in the complete model, although such a mechanism can be incorporated if needed. In the default case we find the nearest-consistent complete model(s) to a given partial model.

If a solution is not found the ALLOY solver returns a *no solution found exception* to AToM^3 (the invoker). We show this result in a dialog box in the AToM^3 environment. In our work we do not debug a partial model to find the exact source of inconsistency. This incurs a computational cost and time as we need to check every partial model predicate expression against the metamodel constraints to see which characteristics of the partial model leads to an inconsistency. We leave it to the user and depend on his/her expertise of the DSML to identify the inconsistent part of the partial model and correct it.

4.2.9 Examples in Completion

In this section, we consider four examples of partial models in the FSM modelling language. The examples are shown in Figure 4.9 (a), 4.9 (b), 4.9 (c) , 4.9 (d) respectively. The synthesized predicates for these models are shown in Figures 4.9 (e), 4.9 (f) and 4.9 (g) , 4.9 (h). The example in Figure 4.9 (a) contains only one State object with none of the properties having been set. The example in Figure 4.9 (b) contains two State objects and a Transition object not connected. In Figure 4.9 (c) we consider a more complex model with several State and Transition objects with some properties set and some not. Finally, in Figure 4.9 (d) we present a model containing at least two State objects with *isInitial* set to True.

We perform the model completion of these models using two methods of setting parameters for completion:



Synthesized Predicates

```
fact partialModel_Fact1
{
  some State
}
```

(e)

Synthesized Predicates

```
pred partialModel2_Fact
{
some s1 : State, s2 : State, t1 : Transition | s1 != s2 and
t1 in s1.outgoingTransition and t1 in s2.incomingTransition
}
```

(f)

Synthesized Predicates

```
pred partialModel3_Fact
{
some s1 : State, s2 : State, s3 : State, s4 : State, t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and s1 != s4 and s2 != s4 and
 t1 != t2 and t1 in s2.incomingTransition and t2 in s3.incomingTransition and
t1 in s1.outgoingTransition  and t2 in s1.outgoingTransition and
s2.isInitial = True and s4.isFinal = True
}
```

(g)

Synthesized Predicates

```
pred partialModel4_Fact
{
some s1 : State, s2 : State, s3 : State, s4 : State, t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and s1 != s4 and s2 != s4
and t1 != t2 and t1 in s2.incomingTransition and t2 in s3.incomingTransition
and t1 in s1.outgoingTransition  and t1 in s1.outgoingTransition and
s2.isInitial = True and s3.isInitial = True
}
```

(h)

Figure 4.9: (a) Partial model 1, (b) Partial model 2, (c) Partial model 3, (d) Partial model 4, (e) Predicate synthesized for Partial model 1 (f) Predicate synthesized for Partial model 2, (g) Predicate synthesized for Partial model 3, (f) Predicate synthesized for Partial model 4

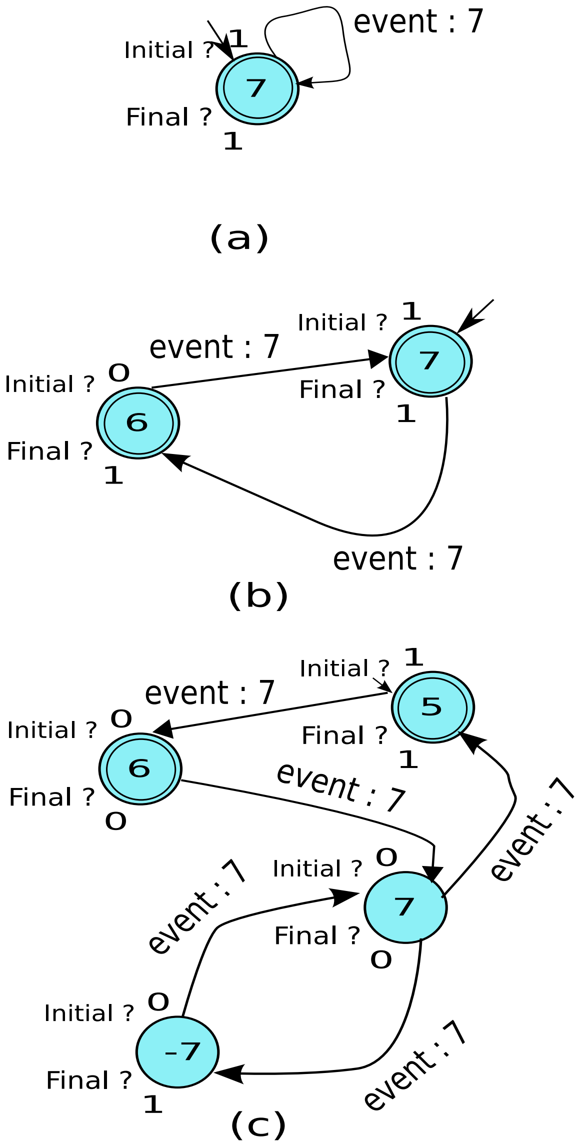


Figure 4.10: (a) Complete Model for Partial Model 1 (b) Complete Model for Partial Model 2 (c) Complete Model for Partial Model 3

- *Scope* : Here we specify a scope as a model completion parameter. The scope is a unique number that defines the maximum number of objects for all concepts in the metamodel. We choose the default scope to be 10. The corresponding ALLOY run statement generated is:

  ```
  pred partialModel {}
  run partialModel for 10
  ```

  The *partialModel* predicate is empty and is simply used to obtain a complete model instance. We solve for up to a scope of 10 objects for each concept in the metamodel.

- *Exact Number and/or Scope* : Another mechanism to complete a model is to specify the exact number of objects and/or scope for objects we expect in the complete model.

  ```
  pred partialModel {}
  run partialModel for exactly 1 FSM, exactly 5 State,
  exactly 10 Transition, 5 int
  ```

  Here we find a solution for a partial model containing exactly 1 FSM object, exactly 5 State objects, exactly 10 Transition objects. Finally we set a bit-width for integers which is 5. This means that all integers range between -2^5 to 2^5.

All the above parameters were initially set in the synthesized AToM^3 modelling environment. The user is only exposed to the graphical syntax of the concepts in the metamodel and with a text-box to specify the exact number of objects or a scope. The model completions were performed on a Macbook Pro laptop with an Intel Core 2 Duo processor running at 2.6 GHz clock speed and with 2 GB of RAM. We use the ALLOY analyzer API to invoke the SAT solver Minisat [111] [112] from Chalmers University to solve the Boolean CNF synthesized from the ALLOY model. The time to obtain the solutions for the four partial models for the completion parameters is presented in Table 4.6.

We show the complete models themselves in Figure 4.10 with a scope of 10. Normally, there is more than one solution to a model completion. We show one of the possible solutions. We do not show that the complete models synthesized for the exact number of objects due to large size of the models. However, it is interesting to note in Table 4.6 that the time taken to synthesize models with the exact number of objects specified for each class is a lot faster even though the models are larger. This is because the additional knowledge about the number of objects makes the search space of the models much smaller, therefore allowing us to obtain a solution faster.

The complete model in Figure 4.10 (a) satisfies all the metamodel constraints such that the single State label has a unique value 7. There is at least one final state and exactly one initial state. In addition, the complete model contains a Transition object of the State to itself with an event 7. This new object added to the complete model does not violate any of the knowledge already present in the partial model.

The second complete model in Figure 4.10 (b) originally was a partial model with two State objects and a Transition object. The complete model now contains two final State objects and

Table 4.6: Model Completion Times

Partial Model	**Description (I=Inconsistent)**	$Time_{Scope}$ (s)	$Time_{Exact}$	$Time_{ScopeScaled}$	$Time_{ExactScaled}$
Fig. 4.9 (a)	Only one State object with no properties specified	1.283	0.447	118.045	32.002
Fig. 4.9 (b)	Two State objects and one Transition object	1.289	0.496	115.994	31.488
Fig. 4.9 (c)	Several State and Transition objects with some properties specified and some not	1.315	0.575	11.4301	32.517
Fig. 4.9 (d)	Several State and Transition objects with two initial State objects	1.291 (I)	0.402 (I)	111.352 (I)	31.734 (I)

exactly one initial State object. There is also an inclusion of a Transition object in the complete model. The synthesized model conforms to all metamodel constraints.

The third complete model in 4.10 (c) contains a complex partial model with additional objects that preserve the knowledge in the partial model. We can scale up to a model with several hundred atoms using ALLOY to obtain results in a reasonable amount of time (for online user interaction with the modelling environment). An atom consists of any non-divisible entity in the ALLOY model. This includes objects and their properties connected via relations.

The fourth partial model in 4.9 (d) consisted of two initial State objects which is not permitted by the metamodel constraint which states that the FSM metamodel must contain only one initial State object. Therefore, the SAT solver was unable to find a complete model that could take into account the partial model.

4.2.10 Conclusion of Model Completion

We present a methodology to synthesize domain-specific model editors with metamodel directed model completion for domain-specific modelling languages. Our goal has been to provide model editors with completion capabilities similar to text or code editors in IDEs such as Eclipse or word processors such as Microsoft Word. A potential future application of our approach is generation of test models from partial knowledge. A DSML user draws a partial test model for testing a model transformation and subsequently sets model completion parameters. Then he/she clicks on a button to generate complete test models that are valid test cases for model transformations. Moreover, the model completions are displayed in the concrete visual syntax of the modelling language while evading all the details in the CNF, XML files, or other intermediate low-level representations. This aspect of our tool helps reduce the time to develop models in the modelling environment as the user only works in his domain language. The user does not need to manually transform his models to a different constraint language, solve his models and return the results to the editor anymore since the underlying model completion process is hidden from the user. After all, the goal of MDE is to leverage modelling to the highest possible level of abstraction.

Our approach uses a modelling language metamodel , the syntax, and its *static semantics* in the form of metamodel constraints to perform model completion. However, since the presented approach is modelling language independent we do not consider *dynamic semantics* often realized in a simulator for model completion. Nevertheless, we project several implications to simulation as it goes hand in hand with modelling. Model simulators, such as MATLAB/Simulink for causal block diagrams, often contain *hard-coded declarative constraints or program statements* that check and report on the validity of input models during simulation. For example, a causal block diagram simulator analyzes input models to detect cycles and warns the modeller. These statements that are integrated in simulator code come from heterogenous sources of knowledge such as domain experience, static/dynamic analysis, and testing. This gradual inclusion of model validity knowledge directly into simulator code makes them bulky and slow to execute. This approach also obscures the user from potentially using this knowledge to build correct models. Extracting knowledge from simulators and developing modelling language invariants to guide modellers to create invariant-validated models leverages a *correct by construction* philosophy. Further, using these invariants for automatic model completion of partial models makes the modelling and simulation process less error-prone as models are first checked and then completed to satisfy invariants before simulation.

Our lightweight approach is effective for small yet useful modelling languages. Time to complete models by the state of the art SAT solvers for about 50 objects in the model is not more than a few minutes for FSM. The completion time greatly *depends on the complexity* of the DSML. The time taken to obtain complete models also gives us insight about how restricted a DSML is and how it can be relaxed.

As future work we intend to run thorough performance experiments on a specific industry strength DSML. Such a DSML will have a larger metamodel with a several complex constraints. We will limit ourselves to the confines of first-order relational logic in ALLOY as the language to express constraints. We also wish to enlist the set of detailed requirements to synthesize DSML modelling environments with completion. For example, an interesting factor is user interaction time. If a complete model is not returned within a given time then the user can no longer make developments quickly. Other aspects of model completion include completion of models when two or more metamodels are involved, expression of partial models as invariants or constraints, and aiding the user by helping him/her set parameters for model completion.

Chapter 5

Automatic Effective Product Discovery

In previous chapters 3 and 4, we have seen how models can be discovered in any modelling domain specified by a metamodel. The generic approach of generating models can be applied to any metamodel. However, not all software systems can be economically modelled and consequently discovered in a modelling language due to existence of reliable software assets. Often, such time-tested legacy software assets are economically viable only in their original form instead of being represented as a model instance in a new modelling language. For instance, the redevelopment / remodeling of the Linux kernel 2.6.25 is estimated to cost of a whopping 1.3 billion dollars! Therefore, we ask how do we discover useful combinations existing software assets to create software ? To answer this question we present automatic discovery in a modelling domain representing the variability in combining existing software assets. The variability in combining different software assets in a software system gives rise to a family of software products called a Software Product Line (SPL). The *feature diagram* (FD) or feature model is a widely used language to specify the modelling domain of a SPL. Elements in the domain of the SPL are called *products* which are obtained by composing configurations of various software assets. In this chapter, we present a methodology and tool AVISHKAR for *automatic discovery of test products* in the modelling domain of a Software Product Line.

The remainder of the chapter is structured as follows: Section 5.1 we introduce automatic effective product discovery. In this thesis, we focus on the specific case of test product discovery in a SPL. The context and the problem for test product discovery is presented in Section 5.2. In Section 5.3, we describe metrics to assess SPL test generation/discovery strategies. Section 5.4 gives an overview of the test product generation methodology and tool AVISHKAR. In Section 5.5 we present two “divide-and-compose” strategies that help scale product generation to large SPLs. In Section 5.6 we present experiments to qualify our strategies on transaction processing SPL case study: AspectOPTIMA. Section 5.7 draws some conclusions and outlines future work.

5.1 Introduction

The idea of automatic effective product discovery in a SPL is illustrated in Figure 5.1. As illustrated in the figure, a feature diagram FD specifies the modelling domain for a SPL. The modelling domain consists of a set of products. **Heterogenous sources of knowledge** may

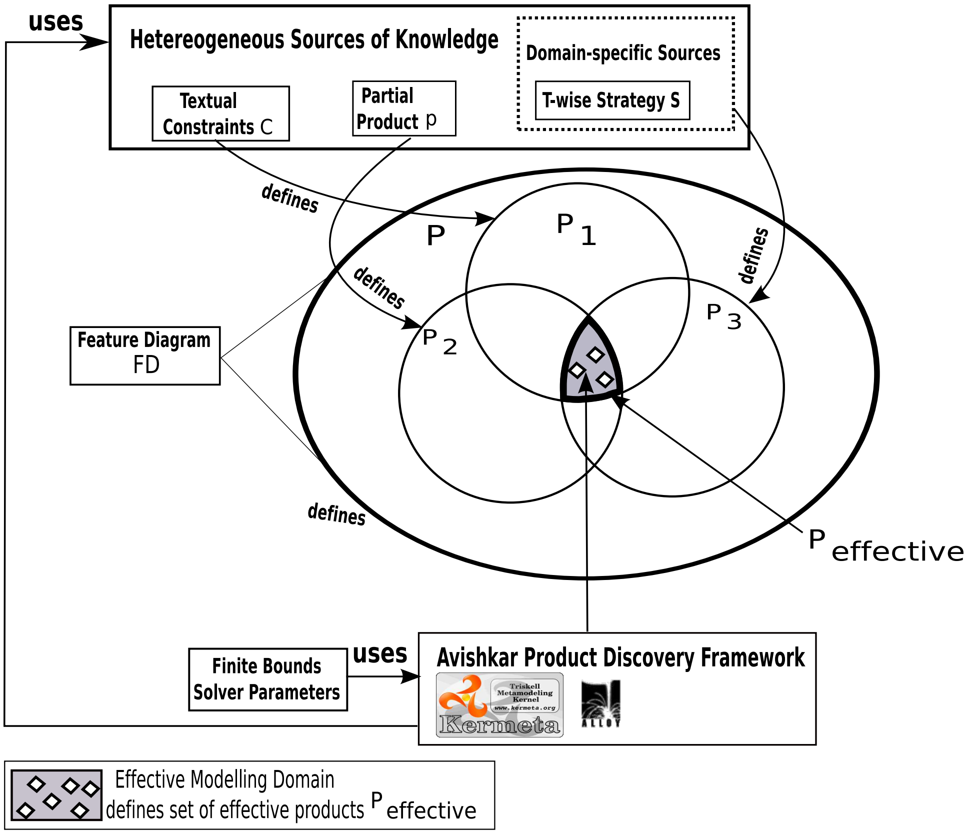

uses
Hetereogeneous Sources of Knowledge
Domain-specific Sources
Textual Constraints C
Partial Product p
T-wise Strategy S
defines
P
P 1
defines
defines
P 2
P 3
Feature Diagram FD
defines
P effective
Finite Bounds Solver Parameters
uses
Avishkar Product Discovery Framework
Kermeta
Effective Modelling Domain
defines set of effective products P effective

Figure 5.1: Automatic Effective Product Discovery

further constrain the modelling domain specified by a feature diagram:

- **Textual Constraints** C are expressed on a set of features. Boolean dependency constraints are expressed textually when they cannot be directly encoded in the FD. These constraints specify the subset $P_1 \subset P$

- **Partial Product** p is a set of features chosen in product. The set of features may require the selection of other features to derive a complete product. The partial product specifies the subset $P_2 \subset P$

- **T-wise Strategy** S is a product generation strategy to detect faults in software product lines [90] [120]. The large number of products specified by a feature diagram can be sampled using a strategy such as $T - wise$. The objective is to generate a minimum number of products that satisfy all $T - wise$ interactions between features. The $T - wise$ strategy for a particular value of T specifies the subset $P_3 \subset P$.

The intersection of all the sources of knowledge defines the *effective modelling domain*. The effective modelling domain is the set of products defined by $P_{effective} \leftarrow P \cap P_1 \cap P_2 \cap P_3$. Can we automatically generate or discover models in the effective modelling domain of products? This is the *general question* that intrigues us.

In this thesis, we address this question for the specific problem of **test generation for software product lines**. Our solution is embodied in the tool AVISHKAR as shown in the Figure 5.1.

Product line testing consists in deriving a set of products and in testing each product. This raises two major issues: 1) the explosion in the number possible products; 2) the generation of test suites for products. The first issue rises from the combinatorial growth in the number of products with the number of features in a feature diagram. In realistic cases, the number of possible products is too large for exhaustive testing. Therefore, the challenge is to select a relevant subset of products for testing. The second issue is to generate test inputs for testing each of the selected product. This can been seen as applying conventional testing techniques while exploiting the commonalities between products to reduce the testing effort [152, 150, 100]. Here, we focus on the first issue: *How can we efficiently select a subset of products for product line testing?*

Previous work [39, 90] has identified combinatorial interaction testing (CIT) as a relevant approach to reduce the number of products for testing. CIT is a systematic approach for sampling large domains of test data. It is based on the observation that most of the faults are triggered by interactions between a small numbers of variables. This has led to the definition of pairwise (or 2-wise) testing. This technique selects the set of all combinations so that all possible pairs of variable values are included in the set of test data. Pairwise testing has been generalized to T-wise testing which samples the input domain to cover all T-wise combinations. In the context of SPL testing, this consists of selecting the minimal set of products in which all T-wise feature interactions occur at least once.

Current algorithms for automatic generation of T-wise test data sets have a limited support in the presence of dependencies/constraints between variables. This prevents the application of these algorithms in the context of software product lines since feature diagrams define complex dependencies between variables that cannot be ignored during product derivation. Previous work [42, 41] propose the use of constraint solvers in order to deal with this issue. However, they still leave two open problems: *scalability* and the need for a *formalism* to express feature diagrams. The former is related to the limitations of constraint solvers when the number of variables and clauses increases. Above a certain limit, solvers cannot find a solution, which makes the approach infeasible in practice. The latter problem is related to the engineering of SPLs. Designers build feature diagrams using editors for a dedicated formalism. On the other hand, constraint solvers manipulate clauses, usually in Boolean Conjunctive Normal Form (CNF). Both formalisms are radically different in their expressiveness and modeling intention. This is a major barrier for the generation of T-wise configurations from feature diagrams.

We propose an approach for automatic discovery/generation of test products that contain all valid t-wise interactions between features. The general approach is to transform the input feature diagram and t-wise interactions to a constraint satisfaction problem followed by solving it. The result is a test products that satisfy the FD and t-wise criteria. However, for large feature diagrams with several dependencies the generation of t-wise products is highly limited by the solver. Current constraint solvers have a limit in the number of clauses ,emerging from FD and

t-wise criteria constraints, they can solve at once. It is necessary to divide the set of clauses into solvable subsets. We compose the solutions in the subsets to obtain a global set. In this work, we investigate two "divide-and-compose" strategies to divide the problem of T-wise generation for a feature diagram into several sub problems that can be solved automatically. The solution to each sub-problem is a set of products that cover some T-wise interactions. The union of these sets cover all interactions, thus satisfying the T-wise criterion on the feature diagram. However "divide-and-compose" strategies may yield a higher number of products to be tested and redundancy amongst them which is the price for scalability. We define metrics to compare the quality of these strategies and apply them on a concrete case study.

Our T-wise testing toolset, AVISHKAR, first transforms a given feature diagram and its interactions into a set of constraints into Alloy [72, 71], a formal modeling language, based on first-order logic, and suited for *automatic instance generation*. Then it complements the Alloy model with the definition of the T-wise criteria and applies one of the chosen strategies to produce a suite of products forming test cases. Finally, metrics are computed giving important information on the quality of the test suite. We extensively applied our toolset on AspectOPTIMA [86, 87] a concrete aspect-oriented SPL devoted to transactional management.

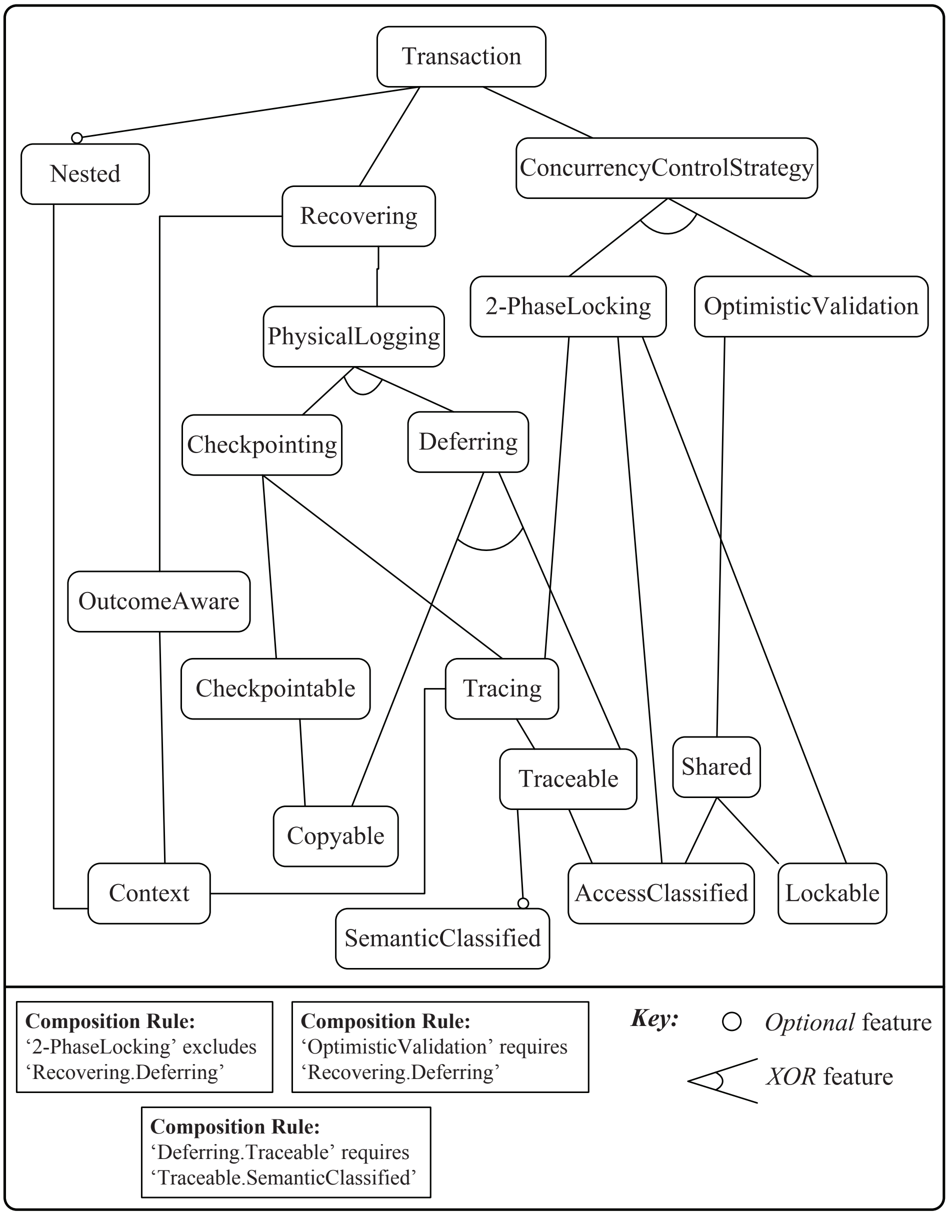
5.2 Context and Problem

In this chapter, we focus on generating a small set of test products for a feature diagram. A product is a valid configuration of the feature diagram that can be used as a relevant test case for the SPL. We give a brief definition and an example of feature diagrams before describing test case generation for them.

Feature Diagram

Feature Diagrams (FD) introduced by Kang et al. [77] compactly represent (Figure 5.2) all the products of an SPL in terms of features [1] which can be composed. Feature diagrams have been formalized to perform SPL analysis [18, 135, 137, 45]. In [135, 137], Schobbens et al. propose an generic formal definition of FD which subsumes many existing FD dialects. FDs are defined in terms of a parametric structure whose parameters serve to characterize each FD notation variant. *GT* (Graph Type) is a boolean parameter indicates whether the considered notation is a Direct Acyclic Graph (DAG) or a tree. NT (Node Type) is the set of boolean operators available for this FD notation. These operators are of the form op_k with $k \in \mathbb{N}$ denoting the number of children nodes on which they apply to. Considered operators are and_k (mandatory nodes), xor_k (alternative nodes) or_k (true if any of its child nodes is selected), opt_k (optional nodes). Finally $vp(i..j)_k$ ($i \in \mathbb{N}$ and $j \in \mathbb{N} \cup *$) is true if at least i and at most j of its k nodes are selected. Existing other boolean operators can usually be expressed with vp. *GCT* (Graphical Constraint Type) is the set of binary boolean functions that can be expressed graphically. A typical example is the "requires" between two features. Finally, TCL (Textual Constraint Language) tells if and how we can specify boolean constraints amongst nodes. A FD is defined as follows:

[1] Defined by Pamela Zave as "An increment in functionality". See `http://www.research.att.com/~pamela/faq.html`



Composition Rule:
‘2-PhaseLocking’ excludes
‘Recovering.Deferring’

Composition Rule:
‘OptimisticValidation’ requires
‘Recovering.Deferring’

Composition Rule:
‘Deferring.Traceable’ requires
‘Traceable.SemanticClassified’

Figure 5.2: Feature Diagram of AspectOPTIMA

- A set of nodes N, which is further decomposed into a set of primitive nodes P (which have a direct interest for the product). Other nodes are used for decomposition purposes. A special root node, r represents the top of the decomposition,
- A function $\lambda : N \mapsto NT$ that labels each node with a boolean operator,
- A set $DE \in N \times N$ of decomposition edges. As FDs are directed, node $n1, n2 \in N, (n1, n2) \in DE$ will be noted $n1 \rightarrow n2$ where n1 is the *parent* and n2 the *child*,
- A set $CE \in N \times GCT \times N$ of constraint edges,
- A set $\phi \in TCL$

A FD has also some well-formedness rules to be valid: only root (r) has no parent; a FD is acyclic; if GT = true the graph is a tree; the arity of boolean operators must be respected. We build upon this formalization to create feature modeling environments supporting product derivation [119] where we encode the AspectOPTIMA SPL feature diagram (see figure 5.2). We implement AspectOPTIMA SPL as an aspect-oriented framework providing run-time support for different transaction models. AspectOPTIMA has been proposed in [87, 86] as an independent case study to evaluate aspect-oriented software development approaches, in particular aspect-oriented modeling techniques. Once we defined the FD, we can create products (i.e a selection of features in the FD). To be *valid*, a product follows these rules: 1) The root feature has to be in the selection, 2) The selection should evaluate to true for all operators referencing them, 3) All contraints (graphical and textual) must be satisfied 4) For any feature that is not the root, its parent(s) have to be in the selection. We enforce the validity of a product according to well-formedness rules defined on our generic metamodel [119] which are automatically translated to Alloy by our FeatureDiagram2Alloy transofrmation (see Section 5.4).

Once we introduce the notion of feature diagram and formalize it we can form our notion of SPL testing on such an entity.

SPL Test Case

A *SPL test case* is one valid product (i.e. a) of the product line. Once this test case is generated from a feature diagram, its behaviour has to be tested.

SPL Test Suite

A *SPL Test Suite* is a set of SPL test cases.

Example

Figure 5.2 presents 3 test cases, three products which can be derived from the feature model. These three test cases form a test suite.

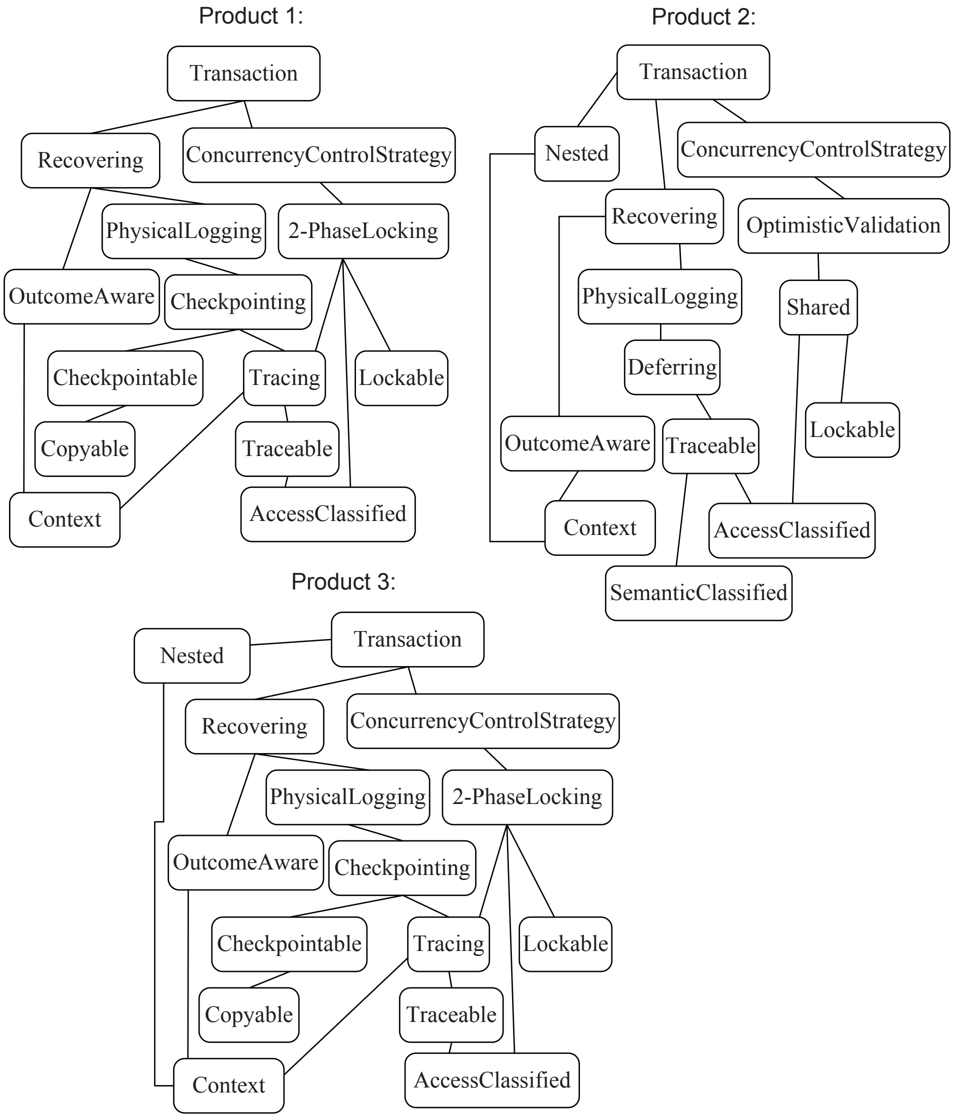


Figure 5.3: Three Test Cases

Valid/Invalid T-tuple

A T-tuple (were T is a natural integer giving the number of features present in the T-tuple[2]) of features is said to be *valid* (respectively *invalid*), if it is possible (respectively impossible) to derive a product that contains the pair (T-tuple) while satisfying the feature diagram's constraints.

Example

In the AspectOptima product line we have a total of 19 features. All these 19 features can take the value true or false. Thus, we can generate 681 pairs that all pariwise combinations of feature values. However, not all of these pairs can be part of a product derivable from the feature model. For example, the pair `<(not Transaction), Recovering>` is invalid with respect to the AspectOptima feature diagram which specifies that the feature `Transaction` is mandatory.

SPL test adequacy criterion

(all-T-tuples): To determine whether a test suite is able to cover the feature model of the SPL , we need to express test adequacy conditions. In particular, we consider the "t-wise" [90, 42] adequacy criteria were each valid T-tuple of features is required to appear in at least one test case.

Example

The test suite presented in figure 5.2 does not satisfy our adequacy criterion since the pair (2-tuple) `<semantic classified, lockable>` does not appear in any of the three test cases.

Test generation

In our context of SPL testing, test generation consists of analyzing a feature diagram in order to generate a test suite that satisfies pairwise coverage.

Pairwise (and more generally t-wise) is a set of constraints over a range of variables (mathematically defined as *covering arrays* [122]). Thus it is possible to use SAT-solving technology [53, 159, 112] to compute such arrays. In our case, variables are the features of a given given feature diagram. It is therefore mandatory to encode a feature diagram in first order logic so SAT-solvers can analyze them. Thanks to feature diagram formalization, this is possible [18, 45] and have been done for various purposes [20, 101].

5.2.1 Problem

The work in this chapter builds upon this idea: model the test generation problem as a set of constraints and ask a constraint solver for solutions. In this context we tackle two issues: (1) modelling the SPL test generation problem in order to use a constraint solver and (2) dealing

[2] In general we will use the term "tuple" to mention a T-tuple when t does not matter. In the special case of pairwise, i.e. when $t = 2$, we denote a 2-tuple by the term "pair".

with the scalability limitations of SAT solvers. Our contribution on the first issue is an automatic transformation from a feature diagram to an Alloy [71] model.

Scalability is a major issue with SAT solvers. It is known that solving a SAT formulae on more than 2 variables in an NP-complete problem. It is also known that depending on the number of variables and the number of clauses, satisfiability or unsatisfiability is more or less computiationally complex [104]. However, we currently don't know how to predict the computation complexity of a given problem. An empirical approach thus consists in trying to solve the set of "constraints all-at-once". Three things can happen: the solver returns a solution, the solver returns an unsatisfiability verdict, the solver crashes because the problem is too complex. In the latter case, one way to generate a test suite that covers t-wise interactions, is to decompose the problem into simpler problems, solve them independently and merge the solutions. In the following, we refer to this approach as "divide-and-compose" approach.

One pragmatic approach, and a naïve one, consists of running the solver once for each T-tuple that as to be covered. This iterative process is the simplest "divide-and-compose" approach and it generates one test case for each valid T-tuple in the FD. For the AspectOPTIMA SPL, we obtain 421 test cases that satisfy pairwise and that corresponds to 421 products to be tested. The all-pairs criterion is satisfied but with a large number of products. It also has to be noted that only 128 different products can be instantiated from the AspectOPTIMA SPL. This indicates that the application of "divide-and-compose", although it might define problems that can be solved, also introduces a large number of redundant test cases in the resulting test suite. Indeed, if it generates 421 test cases, but there can be only 128 different test cases, there is an important redundancy rate.

In general, a solution for generating a test suite with a SAT solver consists in finding a strategy to decompose the SAT problem in smaller problems that can be automatically solved. Also, the strategy should decompose the problem in such a way that when the solutions to all sub-problems are composed, the amount of redundancy in the suite is limited

Test generation strategies

In this chapter we call *strategies* the way we "divide-and-compose". Depending on the strategies and its parameters we will derive more or less test cases. Before delving into the two different strategies we will introduce in the next section metrics to evaluate them.

5.3 Metrics for Strategy Evaluation

We need efficiency and quality attributes in order to evaluate the generated SPL test cases and compare the automatic generation strategies. The first efficiency attribute relates to the size of the generated SPL test suite:

SPL Test suite size

The size of a test suite is defined by the *number of SPL test cases* that a given generation strategy computes. In the best case, we want a strategy to generate the minimal number of test cases to satisfy the SPL test adequacy criterion. As this optimal number is generally not known a priori, we use the SPL test suite size as a relative measure to compare test generation strategies.

A second efficiency attribute relates to the cost of test generation in itself. We measure this cost as the time taken for generation.

SPL strategy time taken

We characterize the cost of a given strategy by the time it took to decompose the problem into solvable sub-problems and the time it took to merge the partial generated solutions to a SPL test suite.

We also evaluate the quality of the generated test cases. First, we want to appreciate the coverage of the generated test cases with respect to the feature diagram. We measure coverage by looking at the rate of similarity between the test cases that are generated. The intuition is that, the more test cases are similar, the less they cover the variety of products that can be generated from the feature diagram.

Test Case Redundancy

We define *test case redundancy* between two valid products as the ratio of *non-compulsory* features they have in common. By *compulsory*, we mean that it comprises mandatory features and features that are explicitly required by them. Put in other terms, for any set of features $F \subseteq N$ representing a *valid* product according to the aforementioned rules for constructing FDs in section 5.2, we form the set $CF \subseteq F$:

$$\begin{aligned} CF = \{\{f_i\} \in N | \forall \{f_j\} \in N \land f_j \mapsto f_i, \\ \forall k \in \mathbb{N}, \lambda(f_j) = and_k \cup \\ \{f_l\} \in N | requires(f_i, f_l) = true \end{aligned}$$

Given a set of feature f_i in all set of feature N in a product, the set CF is the union of the subset of features f_j in N such that a feature f_j is a parent of f_i, or f_j is in a binary AND relation with f_i, and the subset of features f_l such that f_l is required by any f_i. In which *requires* is a binary boolean function (belonging to GCT) such that it returns true if there is a constraint edge labeled as “requires” between theses two features.

Hence the redundancy ratio between two test products p_i and p_j is:

$$r(p_i, p_j) = \frac{card((F_{p_i} - CF_{p_i}) \cap (F_{p_j} - CF_{p_j}))}{card((F_{p_i} - CF_{p_i}) \cup (F_{p_j} - CF_{p_j}))}$$

The sets CF_{p_i} and CF_{p_j} represent the compulsory sets of features for products p_i and p_j while F_{p_i} and F_{p_j} are the sets of all features in products p_i and p_j. This ratio equals to 1 if the two products are the same and 0 if they have no non-compulsory feature in common.

Example

Products 1 and 3 (Figure 5.2) have test case redundancy ratio of 0.88 since they differ only by one feature out of 9 non-compulsory.

At the test suite level, we compute test case redundancy by computing the average of test case redundancy ratio for any two (cartesian product) test cases of the suite.

As a second quality attribute, we want to assess the quality of the generated SPL test cases with respect to T-wise interactions coverage. If we know that, by construction, each tuple appears at least once in the test suite, we also know that the generation process might lead to the repetition of tuples an arbitrary number of times. For the SPL testers, such repetitions imply that they will test the same interaction of features several times.

T-tuple Occurrence

This metric is the number of occurrences of a valid (T-tuple) in a test suite. Let TS be a test suite comprised of p_i valid cases and $F_{p_i} \subseteq N$ be their associated features. Let t a T-tuple ($t = \{f_i \in N\}$). Tuple occurrence redundancy is then:

$$t_o = card(t \in T | t \subseteq F_{p_i})$$

5.4 Test Generation Methodology & AVISHKAR Toolset

In this section, we describe the automatic generation of test products from a feature diagram that satisfy the T-wise SPL test adequacy criteria. Our tool AVISHKAR has been designed to support any value of T. The methodology consists of five key steps shown in Figure 5.4.

The generation is based on ALLOY as the underlying formalism to formally capture all dependencies between features in a feature diagram as well as the the interactions that should be covered by the test cases.

5.4.1 Step 1: Transforming Feature Diagrams to ALLOY

In order to generate valid test products directly from a feature diagram, we need to transform the diagram in a model that captures constraints between features (defined in Section 5.2). The *FeatureDiagram2Alloy* transformation automatically generates an ALLOY model A_F from any feature diagram FD expressed in our generic feature diagram formalism [119].

The A_F model captures all features as ALLOY *signatures* and a set of ALLOY *signatures* that capture all constraints and relationships between features. This model also declares two signatures that are specific to test generation: *configuration* that corresponds to a test case and that encapsulates a set of features (listing 5.2); *ProductConfiguration* (listing 5.3) which will encapsulate a set of test cases.

Example

The AspectOptima feature diagram, shown in Figure 5.2, we have 19 features $f_1, f_2, ..., f_{19}$. The transformation *FeatureDiagram2Alloy* generates 19 signatures to represent these features shown

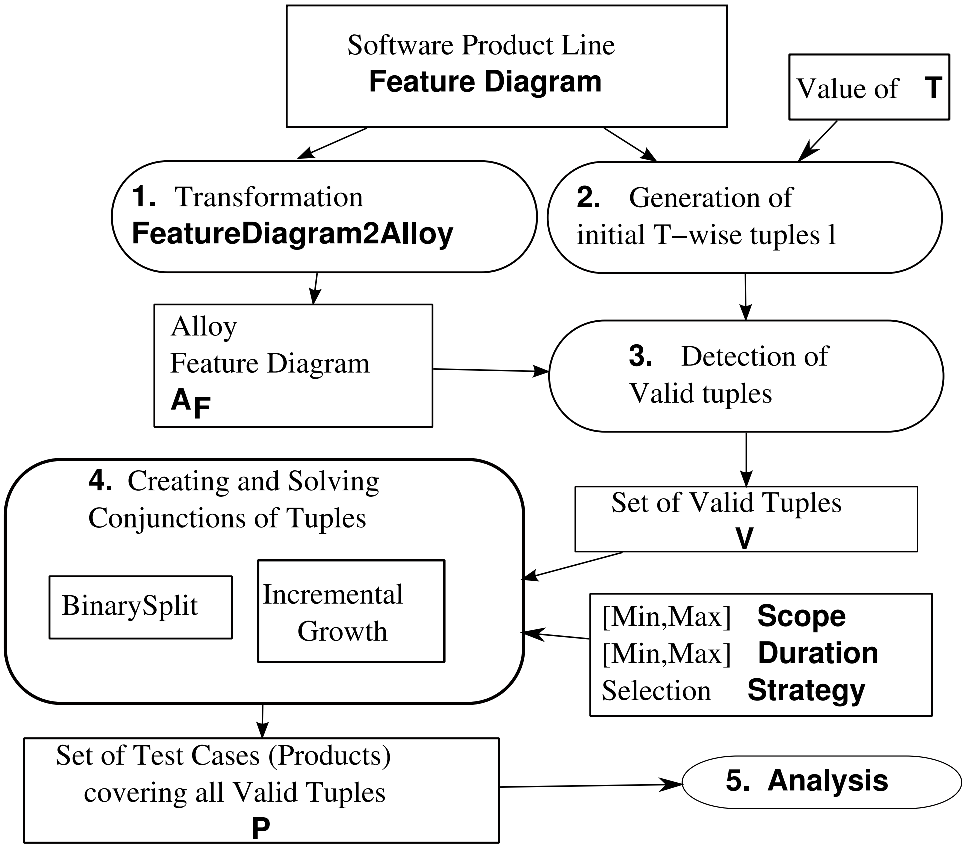


Figure 5.4: Product Line Test Generation Methodology

in listing 5.1. The root feature *Transaction* is always mandatory indicated by the prefix *one* for the field *f* as shown in listing 5.2. Optional features are indicated by the prefix *lone* such as feature *Nested* or *f2* in listing 5.2.

```
sig Transaction {}
sig Nested {}
sig Recovering {}
sig ConncurrencyControlStrategy {}
sig PhysicalLogging {}
sig TwoPhaseLocking {}
sig OptimisticValidation {}
sig Checkpointing {}
sig Deferring {}
sig OutcomeAware {}
sig Checkpointable {}
sig Tracing {}
sig Context {}
sig Copyable {}
sig Traceable {}
sig Shared {}
sig SemanticClassified {}
sig AccessClassified {}
sig Lockable {}
```

Listing 5.1: Generated Signatures for Features in AspectOptima

```
sig Configuration
{
f1: one Transaction, //Mandatory
f2: lone Nested,  //Optional
...
f19: one Lockable //Mandatory
}
```

Listing 5.2: Generated Signature for Configuration of Features in AspectOptima

```
one sig ProductConfigurations
{
  configurations : set Configuration
}
```

Listing 5.3: Generated Signature for Set of Configurations

The *FeatureDiagram2Alloy* transformation generates ALLOY *facts* in A_F.

Example

In listing 5.4, we present two ALLOY facts generated to show the mutually exclusive (XOR) selection of features f_6 (*TwoPhaseLocking*) and f_7 (*OptimisticValidation*) given we select the feature f_4 (*ConcurrencyControlStrategy*). The fact must be true for all configurations.

```

//Two Phase Locking XOR Optimistic Constraint 1
pred TwoPhaseLocking_constraint
{
all c: Configuration |
  #c.f6==1 implies (#c.f4=1 and #c.f7=0)
}


//Two Phase Locking XOR Optimistic Constraint 2
pred OptimisticValidation_constraint
{
all c: Configuration |
  #c.f7==1 implies (#c.f4=1 and #c.f6=0)
}
```

Listing 5.4: Generated Fact for XOR

The *FeatureDiagram2Alloy* transformation has been implemented as a model transformation in the Kermeta metamodeling environment [108]. Since our feature diagram formalism is generic [119, 106] various kinds of feature diagrams can be automatically transformed. We summarize the transformation rules in Figure 5.5. The interpretation of these rules is straightforward. The generated facts in ALLOY state boolean constraints on relevant features in the feature diagram.

5.4.2 Step 2: Generation of Tuples

In Step 2, we automatically compute the set I of all possible tuples of features from feature diagram AF and the number T. The tuples enumerate all T-wise interactions between all selections of features in AF.

Example

The 3-tuple $t =< \#f_1 = 0, \#f_2 = 1, \#f_3 = 1 >$ for the value $T = 3$ contains 3 features and their valuations. In the tuple we state that the set of test products must contain at least one test case that has features f_2 and f_3 and does not have f1.

The initial set of tuples I is the set of tuples that cover all combinations of T features taken at a time. For example, if there are N features then the size of I is ${}_{2N}C_T$ minus all tuples with repetitions of the same selected feature. Each tuple t in I also has an ALLOY predicate representation. An ALLOY predicate representation of a tuple t is $t.predicate$.

The tuple $t =< \#f_1 = 0, \#f_2 = 1, \#f_3 = 1 >$ is shown in listing 5.5.

```
pred t
{
  some c: Configuration | #c.f1=0 and #c.f2=1 and #c.f3=1
}
```

Listing 5.5: Example Tuple Predicate

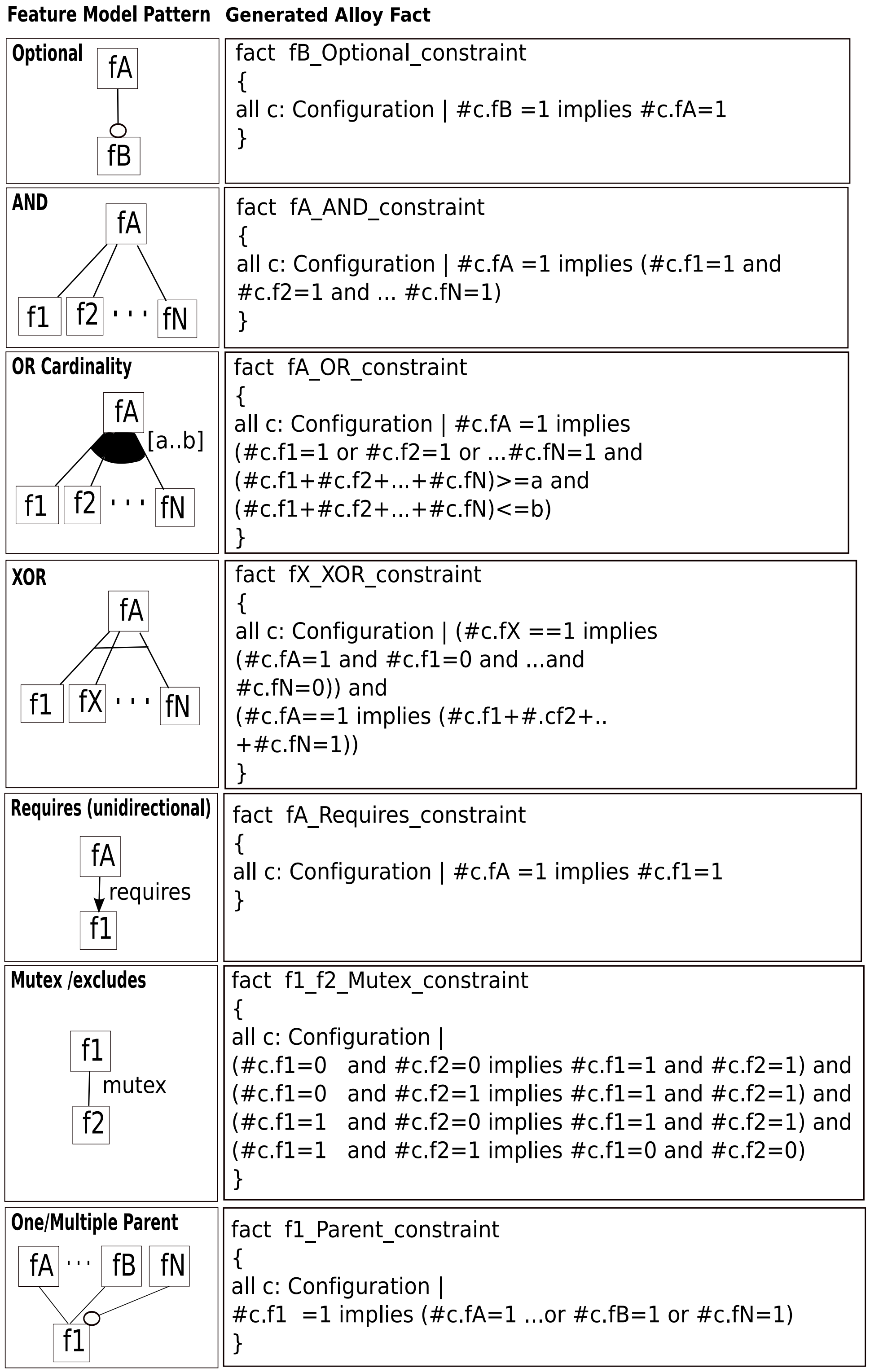
5.4.3 Step 3: Detection of Valid Tuples

In this third step, we use the predicates derived from each possible tuple in order to select the valid ones according to the feature diagram. We say that a tuple is valid if it can be present in a valid instance of the feature diagram F.

Example

Consider AspectOptima (in Figure 5.2) features f_1:Transaction, f_2:Nested, and f_4:ConcurrencyControlStrategy, The 3-tuple $t =< \#f_1 = 0, \#f_2 = 1\#f_4 = 1 >$ is not a valid tuple as the feature f_4 required the existence of feature f_1 and hence we neglect it. On the other hand, the 3-tuple $t =< \#f_1 = 1, \#f_2 = 0, \#f_4 = 1 >$ is valid since all feature selections hold true for F. We determine the validity of each such tuple t by solving $A_F \cup t.predicate$ for a scope of exactly 1. This translates to solving the ALLOY model to obtain *exactly one product* for which the tuple t holds true.

For the AspectOptima case study we generate 681 tuples for pair-wise ($T = 2$) interactions in the initial set I. We select 421 valid tuples in the set V.



Feature Model Pattern | **Generated Alloy Fact**

Optional

```
fact  fB_Optional_constraint
{
all c: Configuration | #c.fB =1 implies #c.fA=1
}
```

AND

```
fact  fA_AND_constraint
{
all c: Configuration | #c.fA =1 implies (#c.f1=1 and
#c.f2=1 and ... #c.fN=1)
}
```

OR Cardinality

```
fact  fA_OR_constraint
{
all c: Configuration | #c.fA =1 implies
(#c.f1=1 or #c.f2=1 or ...#c.fN=1 and
(#c.f1+#c.f2+...+#c.fN)>=a and
(#c.f1+#c.f2+...+#c.fN)<=b)
}
```

XOR

```
fact  fX_XOR_constraint
{
all c: Configuration | (#c.fX ==1 implies
(#c.fA=1 and #c.f1=0 and ...and
#c.fN=0)) and
(#c.fA==1 implies (#c.f1+#.cf2+..
+#c.fN=1))
}
```

Requires (unidirectional)

```
fact  fA_Requires_constraint
{
all c: Configuration | #c.fA =1 implies #c.f1=1
}
```

Mutex /excludes

```
fact  f1_f2_Mutex_constraint
{
all c: Configuration |
(#c.f1=0   and #c.f2=0 implies #c.f1=1 and #c.f2=1) and
(#c.f1=0   and #c.f2=1 implies #c.f1=1 and #c.f2=1) and
(#c.f1=1   and #c.f2=0 implies #c.f1=1 and #c.f2=1) and
(#c.f1=1   and #c.f2=1 implies #c.f1=0 and #c.f2=0)
}
```

One/Multiple Parent

```
fact  f1_Parent_constraint
{
all c: Configuration |
#c.f1  =1 implies (#c.fA=1 ...or #c.fB=1 or #c.fN=1)
}
```

Figure 5.5: Feature Diagram to ALLOY Transformation

5.4.4 Step 4: Creating and Solving Conjunctions of Multiple Tuples

Once we have a set of valid tuples, we can start generating a test suite according to the T-wise SPL adequacy criteria. Intuitively, this consists in combining all valid tuples from V with respect to A_F in order to generate test products that cover all t-wise interactions.

Example

For pair-wise testing in the case of AspectOptima this amounts to solving a conjunction of 421 tuple predicates $t_1.predicate \cap t_2.predicate \cap ... \cap t_{421}.predicate$ for a certain scope. The major issue we tackle in this work is that in general, constraint solvers cannot generate the conjunction of all valid tuples at once.

Using the “all-at-once” strategy on aspectOPTIMA, with 421 valid tuples, the generation process crashes without giving any solution after several minutes using MiniSAT [112] solver.

Hence we derived two “divide-and-compose” strategies to break down the problem of solving a conjunction of tuples to smaller subsets of conjunction of tuples. The strategies we present are *Binary Split* and *Incremental Growth*. Each strategy is by parameterized by intervals of values defining the scope of research for each (sub)-conjunction of tuples, the duration in which ALLOY is authorized to solve the conjunction as well as a strategy defining how features are picked in a tuple. We describe these strategies in more detail in section 5.5. The combination of solutions is a test suite TS that covers all tuples.

5.4.5 Step 5: Analysis

In order to assess the suitability of our “divide-and-compose” strategies and compare their ability to generate test suites, we need to compute the metrics defined in section 5.3. We compute for each generated test suite the number of products or test cases, test case and tuple redundancy. We performed extensive experimentation on AspectOPTIMA by generating test suite with different scope and time values. We present consolidated results of these experiments in section 5.6.

5.5 Two strategies for T-wise SPL Test Suite Generation

As mentioned previously, to be scalable we divide the problem of solving tuples into subproblems, i.e. we are creating conjunctions of subsets of tuples. We solve the conjunction of tuples in each of these subsets using the algorithm presented in Section 5.5.1. The first strategy to obtain subsets of tuples, *Binary Split*, is discussed in Section 5.5.2. We present the second strategy, *Incremental Growth*, in Section 5.5.3.

5.5.1 Solving a Conjunction of Tuples

We solve a conjunction of tuples using the Algorithm 2. We combine the Alloy model A_F with a predicate $CT(S).predicate$ representing the conjunction of tuples in the set $S = t_1, t_2, ..., t_L$. We solve the resulting Alloy model *m* using *incremental scoping*. We create a *run* command *c* starting for a scope between the minimum scope *mnSc* and the max scope $mxScope$. We insert the command c into m. A SAT solver such as MiniSAT [112] or ZChaff [159] is used to solve m. We determine the duration $dur = startTime - endTime$ for each scope value. If dur exceeds maximum duration *mxDur* we stop incrementing the scope. The *solve* method returns the *result* of the SAT solving and the corresponding *solution* if a solution exists.

Algorithm 2 solveCT($A_F, S, mnSc, mxSc, mxDur$) : $Boolean, A4Solution$

```
Let current model $m = A_F \cup CT(S).predicate$
$scope \leftarrow mnSc$
$result \leftarrow False$
$dur \leftarrow 0$
while $scope \leq mxSc \wedge dur \leq mxDur$ do
    Let $c =$ “run” $CT(S).name$ for $< scope >$
    $m \leftarrow m \cup c$
    $startTime = currentTime$
    $solution = SATsolve(m)$
    if $solution.isEmpty$ then
        $result \leftarrow False$
        $scope \leftarrow scope + 1$
        Remove command $c$ from $m$
    end if
    if !$solution.isEmpty$ then
        $result \leftarrow True$
        Break While Loop
    end if
    $endTime \leftarrow currentTime$
    $dur \leftarrow endTime - startTime$
end while
Return $\{result, solution\}$
```

5.5.2 Binary Split

The *binary split* algorithm shown in Algorithm 3 is based on splitting the set of all valid tuples V into subsets (halves) until all subsets of tuples are solvable. We first order the set of valid tuples based on the strategy Str. The strategy can be *random* or based on *distance* measure. In this chapter, we consider a random ordering. The *Pool* is set of sets of tuples. Initially, *Pool* contains the entire set of valid tuples V. If each set of tuples $Pool[i], 0 \leq i \leq Pool.size$ in *Pool* is not solvable in the given range of scopes *mnSc* and *mxSc* or within the maximum duration *mxDur* then *result* is *False* for $Pool[i]$. A single value of $result = False$ renders $AllResult = False$. In such a case, we select the *largest set* in $Pool[i]$ and split it into halves $\{H1\}$ and $\{H2\}$. We insert the halves $\{H1\}$ and $\{H2\}$ into $Pool[i]$. The process is repeated until all sets of tuples in *Pool* can be solved given the time limits and $AllResult = True$. In the worst case, binary split convergences with one tuple a set making $Pool.size = V.size$ as all tuples in V are solvable.

5.5.3 Incremental Growth

The *incremental growth* is shown in Algorithm 4. In the algorithm we incrementally build a set of tuples in the conjunction CT and add it to the *Pool*. The *select* function based on a strategy

Algorithm 3 binSplit($A_F, V, mnSc, mxSc, mxDur, Str$)

```
$AllResult \leftarrow True$
$V \leftarrow order(V, Str)$
$Pool \leftarrow \{\{V\}\}$
repeat
    $result \leftarrow False$
    $i \leftarrow 0$
    repeat
        $\{result, Pool[i].solution\}$
        $\leftarrow solve(A_F, Pool[i], mnSc, mxSc, mxDur)$
        $i \leftarrow i + 1$
        $AllResult \leftarrow AllResult \wedge result$
    until $i == Pool.size$
    if $AllResult == False$ then
        $\{L\} = max(Pool)$
        $\{\{H1\}, \{H2\}\} = split(\{L\}, 2)$
        $Pool.add(\{H1\})$
        $Pool.add(\{H2\})$
    end if
until $AllResult = false$
Return $Pool$
```

Str selects a tuple in V and inserts it into CT. The strategy *Str* can be *random* or based on a *distance* measure between tuples. In this paper, we consider only a random strategy for selection. We select and remove a tuple form V and add it to CT until the conjunction cannot be solved anymore ,i.e., $result = False$. We remove the last tuple and put it back into V. We include CT into *Pool*. In every iteration, we initialize a new conjunction of tuples until we obtain sets of tuples in *Pool* that contain all tuples initially in V or when V is empty.

Algorithm 4 incGrow($A_F, V, mnScp, mxScp, mxDur, Str$)

```
$Pool \leftarrow \{\}$
repeat
    $CT \leftarrow \{\}$
    repeat
        $tuple \leftarrow V.select(Str)$
        $CT.add(tuple)$
        $\{result, CT.solution\}$
        $\leftarrow solve(A_F, CT, mnSc, mxSc, mxDur)$
        if $result == False$ then
            $CT.remove(tuple)$
            $V.add(tuple)$
        end if
    until $result == False$
    $Pool.add(CT)$
until $V.isEmpty$
Return $Pool$
```

5.6 Experiments

The objective for our experiments is: To demonstrate the feasibility of “divide-and-compose” strategies (Binary Split and Incremental Growth) and compare their efficiency with respect to test case generation. All experiments are performed on a real-life feature model: AspectOPTIMA. In this section we report and discuss the automatic generation of t-wise test suites for this model.

5.6.1 Experimental Setting

We automatically generate test suites with the two “divide-and-compose” strategies and compare them according to: (a) the number of generated test cases; (b) the number of tuple occurrences in the test suites; (c) the similarity of the products in the generated test suites. For both strategies we have to set the values for two parameters that specify the search space: the scope and the time limit. We vary scope over 5 values: 3, 4, 5, 6, 7; the maximum duration *mxDur* to find a solution for a given conjunction of constraints is fixed at 1600ms. We generate 100 sets of products for each scope giving us a total of 5×100 sets of products for a strategy. The reason we generate 100 solutions is to study the variability in the solutions given that we use *uniform random ordering* in binary split and *random tuple selection* in incremental growth. Therefore, for two strategies we have $2 \times 5 \times 100$ sets of products or test cases. We perform our experiments on a MacBook Pro 2007 laptop with the Intel Core 2 Duo processor and 2GB of RAM.

Before studying the results of our experiments we note that attempting “solving-all-constraints-as-once” does not yield any solutions for the AspectOPTIMA SPL. This is true even for simple feature models such as AspectOPTIMA that does not lead to derivation of billions of products (like industrial product lines). On the other hand, all executions of both “divide-and-compose” strategies generate t-wise test suites. This first observation indicates that these strategies enable the usage of SAT solvers for the automatic generation of t-wise interactions test suites for both simple and potentially complex feature models. This is the first main result of our study.

5.6.2 Number of Products Vs. Scope

In Figure 5.6, we present the number of products generated for different scopes, which corresponds to the number of test cases in a suite. Each box and its whiskers correspond to 100 solutions generated using a strategy for a given scope. On the x-axis we have scope for two strategies : Binary Split represented by *bin_scope* and Incremental Growth represented by *inc_scope*.

For the binary split strategy, the number of products is high for a scope of 3 (average of 50 products), decreases towards a scope of 5 (average 18 products) and increases again towards a scope of 7 (average of 35 products). In our experiments the scope nearest to the minimal number of test cases is 5. For a scope of 7 we ask the solver to create 7 products per subset of tuples (or pairs) while only 5 products suffice for the same set of tuples leading to *more products that satisfy the same set of tuples*. This is true for highly constrained SPLs such as AspectOPTIMA where the total number of products generated does not exceed a couple of hundred. Therefore, fewer products are sufficient to capture all t-wise interactions. For a scope too small such as 3, binary split gives a large number of products. This comes from the coarse-grain splitting (into

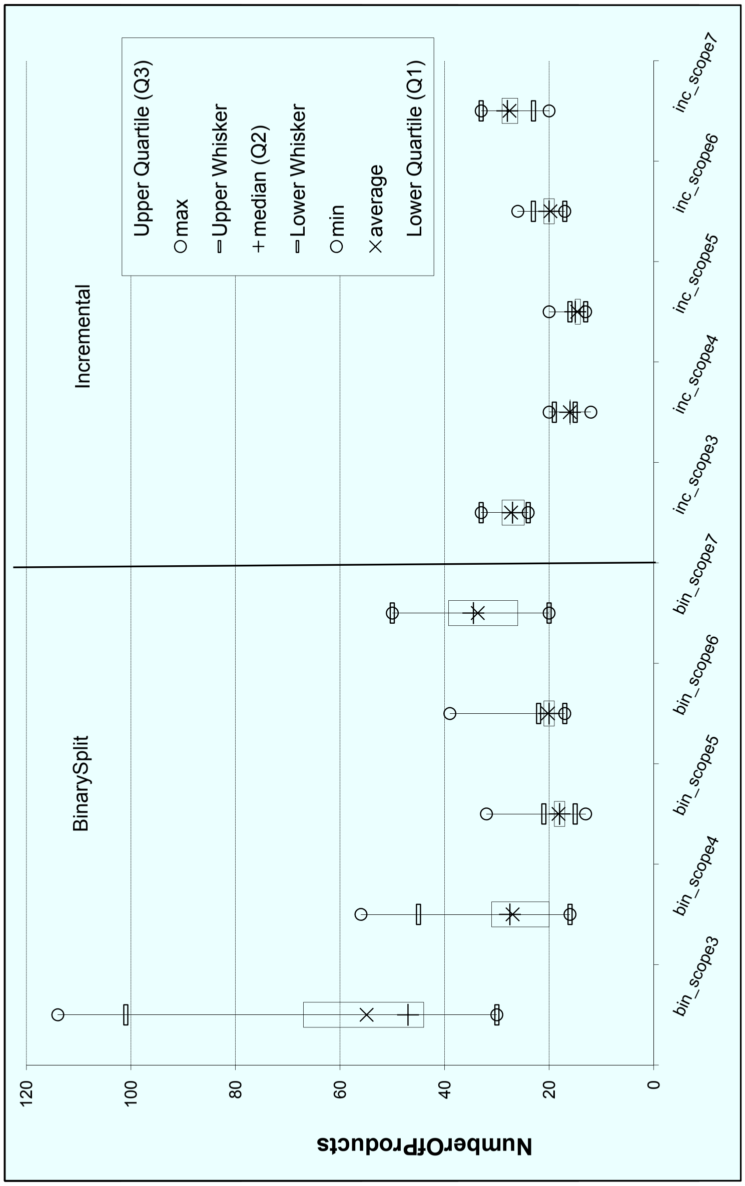


Figure 5.6: Box Plot for Number of Products vs. Scope

halves) of the set of tuples leading to the non-optimal use of 3 products to cover a maximum number of tuples.

For the incremental growth, the general trend that is the high number of products for a scope of 3 (average 25 products), decrease towards a scope of 5 (average 17 products), and increase again towards a scope of 7 (average 27 products). The reasoning for this general trend is similar to binary splitting except that incremental growth attempts to optimize the number of tuples that can be squeezed into a product.

When comparing binary split and incremental growth, there is a notable difference in the variability in the solutions. Binary split results in a large variability (minimum 18 products at scope 5 to a maximum of 115 products at scope 3) in the number of products compared to incremental growth (minimum 16 products to a maximum of 30 products). This is reasonable as binary split applies a coarse-grain strategy of halving sets while incremental growth applies a selective strategy to ’squeeze in’ a maximum number of tuples into a test suite. However, in terms of performance binary split for the AspectOPTIMA case study is far superior compared to incremental growth. Binary split takes an average of 641 ms to obtain a set of products for a scope of 3 while incremental growth takes about 14000 ms. This is primarily due to the fewer steps (average 20) to divide in binary split compared to large number of steps (average 420) for incremental growth. Therefore, we have a trade-off between the size of the test suite and the time to generate the suite. Both strategies are able to automatically find a *small number of test cases* satisfying *all valid pair of feature interactions*.

5.6.3 Tuple Occurrence Vs. Scope

In Figure 5.7, we present a box plot showing the total occurrence of tuples for different scopes. We know that a possible limitation of *divide-and-compose* strategies is that they can generate test cases that cover the same tuple multiple times. This is a limitation for the testing effort, since a redundant tuple means that the same interaction of features has to be tested several times. The total number of valid tuples is 421 for AspectOPTIMA and hence ideally we would like to have a minimum number of products with exactly one occurrence of a tuple. However, the existence of mandatory features force to have multiple occurrences of some tuples in the suite. An effective strategy for test generation is thus a strategy that limits the occurence of the same tuple in the test suite.

For binary split, the total tuple occurrence for a scope of 3 is about 3000 on an average, decreases to about 1400 for a scope of 5 and increases again to 2500 for a scope of 7. Therefore, a scope of 3 generates products with about 7 times the total tuple occurrence compared to the ideal unique occurrence, scope of 5 about 3 times. We again observe that the near-optimal scope of 5 has the least total tuple repetition.

For incremental growth, the total tuple occurrences are lower compared to binary split.

Binary split and scope 3 gives products with 1.6 times more occurrences compared to incremental growth. In general, incremental growth converges to a better set of products: less products with less occurences of tuples.

The strategy and the scope help us choose the ideal set of test cases.

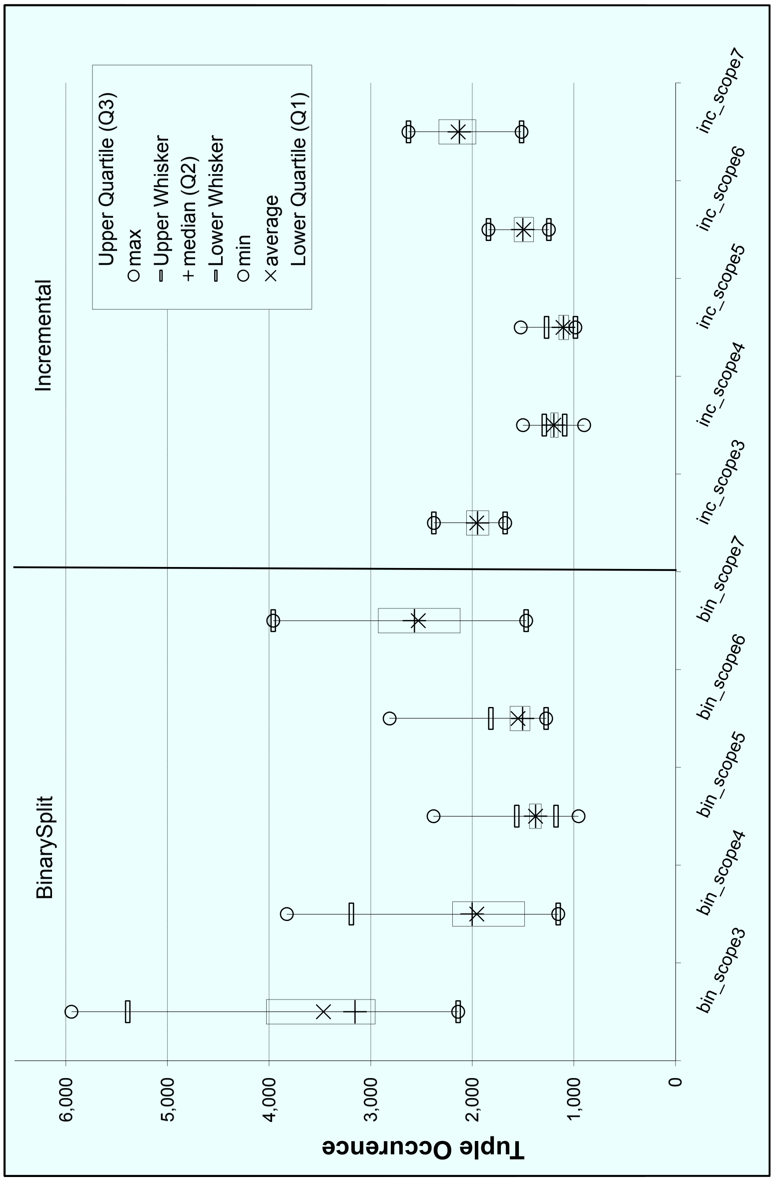


Figure 5.7: Box Plot for Total Tuple Occurrence vs. Scope

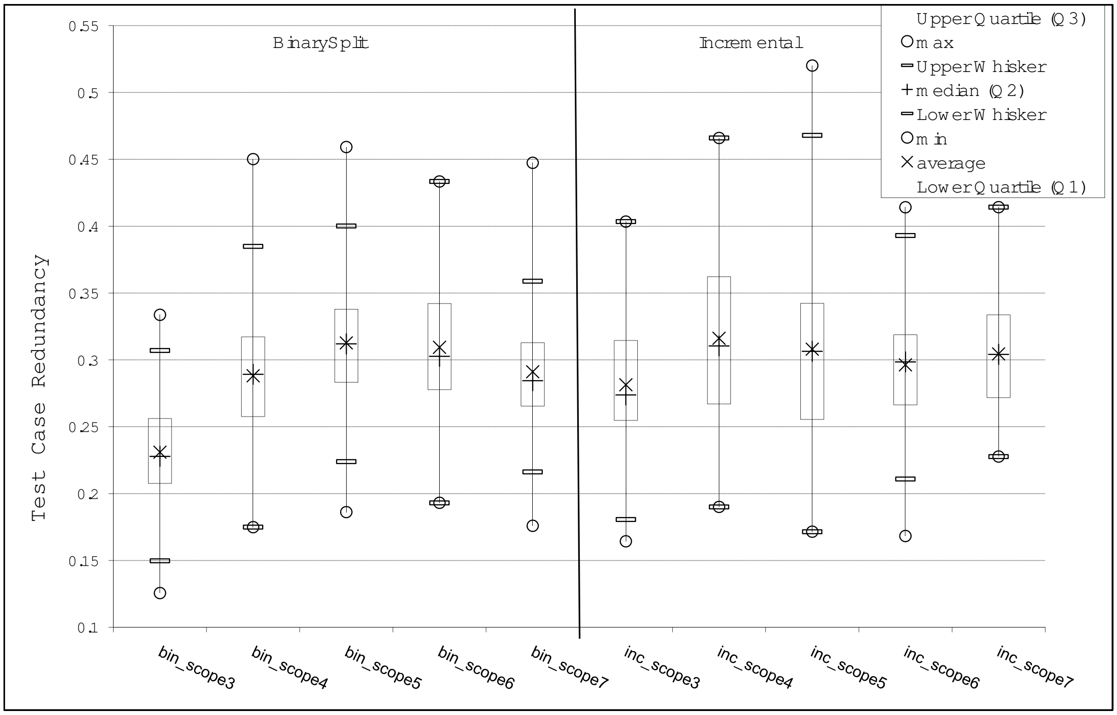


Figure 5.8: Box Plot for Test Case Redundancy

5.6.4 Test Case Redundancy

Results for test case redundancy are presented in Figure 5.8. One first observation is that the values are similar (except for scope 3) for BinarySplit and IncrementalGrowth strategies. This can be because both strategies are based on random ordering of tuples. Hence the coverage of the feature diagram by SPL test cases is quite similar and its particular structure does not influence test case redundancy between the two strategies.

We also observe that test case redundancy increases when the number of products decreases for both strategies, the minimum being obtained with scope 5. This can be explained by the fact that when the number of products decreases, the generator must “fill” each product with more non-compulsory features in order to cover each tuple at least once. When we give more “freedom” to the strategies (by increasing the number of products), they have more options to fill products with non-compulsory features and generate less test case redundancy on average. High redundancy in a small test suite can be beneficial for test cases reuse [152]. However, high redundancy also means similar test cases in a suite and thus less coverage of the SPL, which might not be a good caracteristic of a test suite. ults, which means it has to be tuned for

5.6.5 Threats to Validity

This work mainly focused on the definition of two *divide-and-compose* strategies and the experiment was performed on only one real-world feature diagram. It is a realistic FD, in size and complexity of the constraints between feature. However, since we evaluate our strategies only on this one, there is an important threat to *external validity*. We cannot know how the trends we observed for both strategies can be generalized to feature diagrams with more features or a different topology. We are currently running similar experiments on larger feature models (and less constrained) to assess the impact of topology on the effectiveness of our strategies and implementation. We also have another threat to *construct validity*: we have developed the tools to measure the different metrics on the test suites. Concerning the metrics themselves, they are usual metrics to evaluate test suites (number of test cases, coverage) that we believe are relevant for the evaluation of the proposed strategies.

5.7 Conclusion

In this chapter, we propose an approach and platform supporting the automated generation of test cases for software product lines. Our work is motivated by concerns of scalability and usability. With respect to the first concern, we combined combinatorial interaction testing, as a systematic way to sample a small set of test cases, with two "divide-and-compose" strategies. These strategies address the scalability limitations of SAT solvers used to generate test cases that satisfy all constraints captured in a feature model. Using these strategies, we are able to automatically generate sets of test cases for a medium-sized realistic SPL such as AspectOPTIMA which could not be processed in an "all-constraints-at-once" fashion . We assessed our strategies by computing metrics and discussed the factors that influence test case generation. We addressed usability via model driven engineering techniques [81] to automatically transform generic feature diagrams into alloy models amenable to t-wise test generation in Alloy.

We would like to extend our work along two main dimensions. The first one concerns test generation strategies. We are currently experimenting with toolset on a crisis management system which is characterized by a large number of optional and alternative features inducing more than one hundred billions of possible test cases for exhaustive covering. Using the incremental strategy we were able to reduce this number to a few hundred. We would also like to exploit the feature model structure to reduce the number of tuples to consider and fine-tune t-wise generation. Generated products testability is the second dimension for future work. We would like to extend our test case generation platform with automated SPL derivation techniques such as [119] acting as oracles. This will then form a complete SPL test methodology starting from considering the SPL "as a whole" to individual product testing.

Chapter 6

Conclusion and Perspectives

Model-driven engineering is leveraging the use of models in all several aspects of software development. Research into the theories, techniques, and tools for the various parts that make up a model driven system -models and transformations- is active and is seeing uptake in industrial contexts. However, as MDE is advancing it is facing challenges that characterize software engineering such as managing scalability, reliability and of particular interest in this thesis *automatic discovery of effective models to facilitate test-based validation and model construction*.

In order to address the challenges in *automatic model discovery*, we must develop mechanisms to explore and discover models in a modelling domain. Further, the models must conform to constraints heterogenous sources of knowledge such as metamodel constraints, search strategies, and partial models. How can we discover models in a modelling domain?

We address this question in the thesis by presenting a generic methodology that transforms a modelling domain and heterogeneous sources of knowledge to a constraint satisfaction problem in the formal specification language ALLOY. We solve the constraint satisfaction problem to discover models of interest. We specialize the generic methodology to first consider discovery in a modelling domain specified by a metamodel and constrained by heterogeneous sources of knowledge. This approach is concretely embodied in the tool CARTIER. We validate our approach and CARTIER by performing experiments in test model generation and partial model completion. Second, we specialize our generic methodology for discovery in a modelling domain specified by a feature diagram of a Software Product Line. An SPL allows modelling variability in software systems using legacy software assets. This proves to be better than modelling everything from scratch in a modelling language specified by a metamodel. The methodology is embodied in the tool AVISHKAR. We validate AVISHKAR using experiments to generate test products for a transaction processing SPL AspectOPTIMA. Using both methodologies and tools CARTIER and AVISHKAR we demonstrate the feasibility of automatic model discovery in different modelling domains.

The rest of the chapter is organized as follows. In Section 6.1, we present a summary of the different chapters in this thesis. In Section 6.2, we present ongoing work on the use of AVISHKAR to analyze variation in QoS of web service orchestrations. Finally, in Section 6.3, we present perspectives for future work.

6.1 Summary and Conclusion

Chapter 2, presents the general context of MDE and the creation of modelling domains in MDE. In particular, we discuss (a) The modelling domain specified by a metamodel and its constraints in OCL and (b) The modelling domain specified by a feature diagram. The modelling domain specification are transformed to constraint satisfaction problem (CSP) in the formal specification language ALLOY which we describe in this chapter. The model transformation language to perform the transformation from modelling domain to ALLOY is Kermeta. We describe Kermeta, aspect-weaving in Kermeta, and model typing in Kermeta in this chapter. The chapter presents the state of the art in automatic model discovery with emphasis on test model generation and partial model completion. It also presents the state of the art in automatic product discovery.

In Chapter 3, we present a framework for automatic model discovery in the modelling domain specified by an input metamodel. The framework is embodied in the tool CARTIER. First, we present a metamodel pruning algorithm to extract an effective metamodel from the input metamodel. The effective metamodel is a supertype of the input metamodel from a type-theoretic point of view and a subset of the input metamodel from a set-theoretic point of view. Second, we present a transformation of any metamodel or the effective metamodel to a CSP in ALLOY. The transformation takes into account all non-trivial artifacts in a metamodel such as multiple inheritance, multiplicity, containers, composite properties, opposite properties, and identity properties. A discussion on the validity and complexity of the transformation is presented. Third, we discuss how heterogeneous sources of knowledge such as OCL constraints may be transformed to ALLOY. Finally, we demonstrate the generation of models for the large case study of the UML metamodel.

In Chapter 4, we present experiments to validate automatic model discovery presented in Chapter 3. We present experiments in test model generation and partial model completion in a model editor. First we consider test model generation where we use input domain partitioning strategies to generate test models using CARTIER. These models detect 93% of the bugs in a representative model transformation compared to only 70% for unguided generation. The representative transformation from UML class diagrams to RDBMS models exercises most model transformation operators. The input metamodel UML contains almost all complex metamodel constructs and is a widely used industrial metamodel. In the second experiment we perform partial model completion in a model editor. Given a partially specified model in a model editor we use CARTIER to generate recommendations to complete partial model. We present an algorithm to transform a partial model to an ALLOY predicate. We solve the predicate to generate one or more model completions for models in the Hierarchical Finite State Machine modelling language. We present the different times taken for completion of partial models of various size.

Chapter 5, we present a framework for automatic product discovery in the modelling domain specified by the feature diagram (FD) of a Software Product Line (SPL). The framework is embodied in the tool AVISHKAR. We first transform a FD to a CSP in ALLOY. We solve the resulting ALLOY model to generate products. The focus of this chapter is to discover test products that satisfy the T-wise coverage criteria between features in the FD. Generation of test products for large FDs using ALLOY is not tractable. We scale the use of ALLOY using divide-and-compose strategies that can generate a close to minimal set of test products that satisfy

T-wise coverage. A side-effect of using divide-and-compose strategies is the introduction of redundancies of pairs in products. We presents metrics to measure these redundancies. Using pairwise coverage we show that AVISHKAR generates test products with acceptable redundancy for a transaction processing FD AspectOPTIMA.

6.2 Ongoing Work: Variability Modeling and QoS Analysis of Web Services Orchestrations

In ongoing work we model the variability in a composite web services orchestration using FDs. We apply AVISHKAR to generate different possible orchestrations of a composite web service. We analyze the consequent variation in Quality of Service (QoS) of these orchestrations using probabilistic models of QoS. This work is described below.

Inherent choice in an ever-growing world of services is making *orchestration variability* a significant aspect of a composite web service. The different ways of orchestrating atomic services can be seen as either multiple variants of a composite service created offline or an online composite service that reconfigures dynamically. In either case, we expect to observe variation in Quality of Service (QoS) across different orchestrations. This variation in QoS must not only take into account service variability but also the uncertainty/probabilistic nature of QoS itself.

It is important to consider orchestration variability and its implications on composite service behavior. For instance, not considering variability leads to misrepresentation of contractual agreements on QoS [151]. Contractual agreements such as service level agreements (SLAs) [117] is the industry standard to ensure QoS compliance between service providers and customers. Usual deviations from SLAs are a result of non-incorporation of QoS variability and in particular QoS outliers in its specification. Therefore, we need systematic analysis of variability in order to improve robustness of contractual SLAs.

Modeling variability in web service orchestrations and analyzing the consequent variation in QoS is the principal subject of this work. We present a methodology to model orchestration variability using *feature diagrams* (FDs). Feature diagrams [77] provide a graphical constraints-based framework to specify a product-line of orchestrations. Each orchestration in the product-line is represented as an authorized configuration of invoked/rejected atomic services. In most cases the FD specifies a very large set of configurations making exhaustive sampling infeasible. Instead, we sample the set of all possible configurations by systematically analyzing configurations covering all valid pairwise service interactions [46]. Finally, we use probabilistic models of QoS [129] to analyze variants of orchestrations derived from all valid configurations.

We use our methodology to investigate merits of systematically sampling the set of all configurations of web service orchestrations. Random sampling of configurations, generally employed, is both ineffective and expensive because it cannot be systematic and requires computing QoS values for a large number of configurations. Moreover, random sampling is not easy when FD constraints like mutual exclusion/requirement need to be satisfied. This work focuses on the adaptation of combinatorial interaction testing (CIT) [39] to select a sample of configurations that covers all pairwise interactions of services while satisfying all FD constraints. We use the recently proposed scalable approach in [120] for generating these configurations. CIT is based on the observation that most of the faults are triggered by interactions between a small number of variables [90]. For example, consider the output quality of printing web pages depending on a hypothetical combination of parameters represented in Table 6.1.

Parameters	**Options**
Operating System	Windows, Linux, Macintosh
Browser	IE, Firefox, Chrome, Opera
Printer Model	HP, Canon, Xerox, Epson
Printer Type	Ink-Jet, Laser
Orientation	Portrait, Landscape
Size	A3, A4, A5, A6
Color	B/W, Multicolor

Table 6.1: Examples of printing parameters requiring comparison.

An exhaustive generation of combinations of these parameter options would entail 1536 cases with many redundancies. Pairwise coverage of optional combinations would require just 17 tests, resulting in a reduction of close to 99%. The number of exhaustive tests will increase exponentially with addition of more parameters/options requiring an employment of efficient sampling strategies.

Pairwise coverage test generation has been used to detect faults in software systems in prior work [46], [39]. However, the application of these coverage-based techniques to sample configurations in service orchestrations is yet to be examined. This work performs such an examination through a series of experiments that aim at investigating several facets of the question: is pairwise service interaction sampling of orchestration configurations effective for overall QoS analysis and the consequent definition of a global SLA?

Our experiments are based on a *crisis management system* (CMS) case study described comprehensively in [85]. This work reports on the following questions:

- Is it possible to automatically sample the orchestration configurations space to select configurations that cover all pairwise service interactions?

- What global QoS metrics can we infer from a pairwise sample?

- How stable is the SLA computed from a pairwise sample? This question is related to the fact that the automatic generation of pairwise configurations is not deterministic and thus the global contract might vary depending on the generated *sample*.

- Is pairwise sampling more effective and efficient compared to exhaustive sampling of the configuration space?

From our experimentation, we have seen that analysis of a family of configurations (and their corresponding QoS values) can be accurately represented by a small set of configurations satisfying pairwise interactions. Consistency of various generated pairwise solutions are also demonstrated through simulations. This comprehensive analysis of variability helps the orchestrator understand the global QoS extremities of the composite service before negotiating a SLA agreement. Deterioration in service quality or non-compliance of SLA standards during online deployment of the service is thus prevented. Improvements in the orchestration model to eliminate some deviant configurations (causing excessive deterioration of end-to-end QoS) or grouping a family of configurations with similar QoS behavior are other extensions of this technique.

Accurate offline analysis of a composite web service before its deployment is essential to ensure non-repudiation of a SLA contract. This is necessary to maintain optimal QoS behavior of mission-critical services such as crisis management. In order to do this, the service provider must keep in mind the probabilistic aspect of QoS parameters and the variable configurations in a composite service. In this work, we study an analysis framework to test the QoS of an orchestration before deployment. Further, the notion of systematic pairwise sampling procedure has also been demonstrated, which provides a more efficient sampling of the configuration space than exhaustive trails while still maintaining sufficient coverage. Larger FD and orchestration models can be analyzed using the divide-and-compose approaches [120] to handle this scalability issue. This should provide a simple, systematic and stochastically correct methodology for pre-deployment QoS analysis of a composite service.

While this work concentrates on a particular composition of fixed atomic services, a future area of interest would be optimal compositions. The use of configurations and scenarios modeled by a FD leads to a family of composite services. These, in turn, may be used to generate many versions of the orchestrations. Further implementation of these techniques to study larger composite orchestrations is useful for both obtaining realistic QoS bounds and product generation of families of services.

6.2.1 Related Work

The combinatorial testing framework described by Cohen et al. [39] has been applied extensively to efficient testing for fault detection. In the work of Cohen et al. [40], this technique is extended to software product lines with highly configurable systems. Modeling variability in SPLs using feature models is the work of Jaring and Boschet [74] where they show that the robustness of a SPL architecture is related to the type of variability. To ensure that constraints in the FD are incorporated in the efficient sampling of t-wise tests, the solver proposed by Perrouin et al. [120] is used. In [95] Larsen et al. define modal I/O automata, an extension of interface automata with modality. These allow models of varying configurations to be developed from a single produce line while disallowing trivial implementations. Such a notion when extended to a composite service can provide interesting configurations and versions of composite products as described in [95].

Pre-deployment testing of SLAs has been studied by Di Penta et al. [118], where they make use of genetic algorithms to generate test data causing SLA violations. Analysis of white and black box approaches are provided in the paper. In [31], Bruno et al. make use of regression testing to ensure that an evolving service maintains the functional and QoS assumptions. The service consistency verification due to evolution is done by executing test suites contained in a XML encoded facet attached to the service.

The use of probabilistic QoS and soft contracts was introduced by Rosario et. al [129] and Bistarelli et al. [23]. Instead of using fixed hard bound values for parameters such as response time, the authors proposed a soft contract monitoring approach to model the QoS measurement.

The composite service QoS was modeled using probabilistic processes by Hwang et al. [68] where the authors combine orchestration constructs to derive global probability distributions.

In our work, we extend these two notions to analyze the QoS of a composite orchestration under various configurations. The hard contract notions of end-to-end QoS are replaced by the probability quantile based approach. This provides the service provider the technique for estimating composite service QoS distributions and estimating the global soft contract SLA. Though formal analysis of end-to-end QoS has been studied in Cardoso et al. [35], there are no practical testing tools available for the service provider. The pairwise testing procedure has been shown to outperform other testing techniques in [39]. We extend this testing tool to develop a generic testing methodology to query end-to-end QoS of a web service. The efficacy of this scheme is provided though experimental verification.

Related empirical studies of optimal QoS compositions make use of genetic programming in Canfora et al. [34] and linear programming in Zeng et al. [161]. These are dynamic techniques to choose the best possible atomic services and configurations keeping QoS in mind. This differs from our work due to the assumption that the atomic services and their composition have already been defined. The goal is to analyze the variable configurations that may result due to invocation or non-invocation of particular web services. This is of need when atomic SLAs and their interactions in an orchestration have already been established. Such efficient, systematic and stochastically correct analysis provides an accurate estimate of the global QoS distributions of composite services.

6.3 Perspectives

The ideas presented in this thesis represents a first step towards automating discovery of models in a modelling domain. The work evokes a number of future avenues of research.

6.3.1 A Family of Metamodel Pruning Algorithms

In Chapter 3, we present the metamodel pruning algorithm to extract an effective metamodel from an input metamodel. We show that the effective metamodel is a supertype of the input metamodel from a type-theoretic point of view. It is also a subset of the input metamodel from a set-theoretic point of view. The supertype property of the effective metamodel makes it *backward compatible* with the input metamodel. By backward compatibility we mean all model transformations or operations for the effective metamodel are valid for the input metamodel. Similarly, all models of the effective metamodel are also valid instances of the input metamodel. This property has practical implications to the usage of large industry standard metamodels such as the UML. Experts may extract a small and relevant subset of the UML to create models or transformations while preserving type conformance with UML itself. Therefore, the type conformance property between an effective metamodel and the large input metamodel leverages several applications of the metamodel pruning algorithm. In future work, we would like to investigate the possibility of creating a family of metamodel pruning algorithms.

The notion of a family metamodel pruning algorithms is based on the possibility of developing combinations of atomic pruning operators that satisfy type conformance. An atomic pruning operator has an input metamodel and gives an effective metamodel as output. The effective metamodel shows type conformance with the input metamodel. A given sequence of pruning operators on an input metamodel should give an effective metamodel as output such that it shows type conformance with the original input metamodel. This is due to a transitivity property of pruning operators in a sequence. What are the different possible sequences of pruning operators? Which pruning operators are commutative? Which pruning operators in sequence show transitivity? These are some of the questions that need explorations.

6.3.2 Transforming OCL Subset to ALLOY

In Chapter 3, we present a complete transformation of a metamodel to ALLOY implemented in the CARTIER framework. However, not all constraints may be expressed in the metamodel. A textual constraint language such as the Object Constraint Language (OCL) is the industrial standard to expressed additional metamodel constraints. OCL is a side-effect language that queries a model of a modelling language and check structural properties on the model. There are several similarities between OCL and ALLOY in the way constraints are expressed. In future work, we would like to focus on transforming a subset of OCL to ALLOY facts or predicates. ALLOY also has some features not yet exploited in OCL which may help concurrently improve OCL itself. In [155], the authors presents some shortcomings of OCL with respect to ALLOY.

6.3.3 Product Discovery Strategies based on Feature Diagram structure

In Chapter 5, we present the AVISHKAR framework to generate products that satisfy all T-wise feature interactions in a FD. We believe that the quality of the test products and the number of effective test products may be improved if we consider the structural semantics of the FD in developing new strategies. New strategies will essentially comprise of analyzing the tree structure of the FD to obtain knowledge to generate test products. The idea is to generate test products using knowledge that explore the FD's product space while respecting FD constraints. This is in contrast, to T-wise generation where a lot of feature interactions are generated that do not satisfy the FD constraints. Only a subset of the T-wise interactions are valid and are used to generate test products.

6.3.4 Scaling Constraint Solving using ALLOY

In most of the thesis we have used ALLOY to generate models or products. Generation using ALLOY is based on the hypothesis that small models are often effective. We demonstrate this using experiments in test model generation. However, for product generation we make advances in scaling ALLOY to generate products for a large FD. The idea is based on dividing the constraint satisfaction problem and composing the results into a final set of products. This approximate approach can handle large FDs but introduces some tuple redundancy in the generated products. What are other ways to scale the size and number of models that can be generated using ALLOY ? This is a question that intrigues us. We would like to research this question in two axes: (a) Develop divide and compose strategies to first create small models and then weave them together into larger models (b) Leverage SAT solving using parallel SAT solvers such as ManySAT [66] in order to generate instances from a large and highly-constrained ALLOY model.

Appendix

6.4 ALLOY Model of UMLCD Synthesized by CARTIER

```
module tmp/UMLCD
open util/boolean as Bool

sig Model
{
  classifier : set Classifier,
  association:set Association
}


abstract sig Classifier
{
  name :  Int
}

sig PrimitiveDataType extends Classifier
{ }

sig Class extends Classifier
{
  is_persistent: one Bool,
  general : lone Class,
  attribute : some Property

}

sig Association

{
  name: Int,
  memberEnd: one Class,
  ownedEnd: one Class

}

sig Property
{
  name: Int,
  is_primary : Bool,
  datatype: one Classifier
}

//Meta-model constraints

/* There must be No Cyclic Inheritance in an UMLCD*/

fact noCyclicInheritance
{
  no c: Class | c in c.^general
}

/* All the attributes in a Class must have unique attribute names */

fact uniquePropertyNames
{
  all c:Class | all a1:  c.attribute, a2: c.attribute | a1.name = a2.name implies a1=a2
}

/* An attribute object can be contained by only one class */
```

```
fact attributeContainment
{
  all c1:Class, c2:Class | all a1: c1.attribute , a2 : c2.attribute | a1 = a2 implies c1=c2
}

/* There is exactly one Model object */

fact oneModel
{
  #Model=1
}

/*All Classifier objects are contained in a Model*/

fact classifierContainment
{
  all c:Classifier | c in Model.classifier
}

/*All Association objects are contained in a Model */

fact associationContainment
{
all a:Association| a in Model.association
}

/*A Classifier must have a unique name in the Class Diagram*/

fact uniqueClassifierName
{
  all c1:Classifier, c2:Classifier |c1.name = c2.name implies c1=c2
}

/*An associations have the same name either they are the same association or they have different sources*/

fact uniqeNameAssocSrc
{
  all a1:Association, a2:Association |
  a1.name = a2.name implies (a1 = a2 or a1.src != a2.src)
}
```

Listing 6.1: ALLOY Model for UML Class Diagram

6.5 Initial Set of Pre-conditions

```
/*Initial Model Transformation Pre-conditions*/


fact atleastOnePrimaryProperty
{
  all c:Class | one a:c.attribute | a.is_primary =True
}

fact no4CyclicClassProperty
{
  all a:Property | a.datatype in Class implies all a1:a.datatype.attribute | a1.datatype in
  Class implies all a2:a.datatype.attribute | a2.datatype in Class implies all a3:a.datatype.attribute|a3.datatype
   in Class implies all a4:a.datatype.attribute | a4.datatype in PrimitiveDataType
}

fact noPropertyAndAssociationHaveSameName
{
  all c:Class , assoc :Association |
  all a:c.attribute | (assoc.src = c) implies a.name != assoc.name
}

fact no1CycleNonPersistent
{
all a: Association | (a.memberEnd = a.ownedEnd) implies a.ownedEnd.is_persistent = True
}

fact no2CycleNonPersistent
{
  all a1: Association, a2:Association |
  (a1.memberEnd = a2.ownedEnd and a2.memberEnd = a1.src) implies
  a1.ownedEnd.is_persistent = True or a2.ownedEnd.is_persistent=True
}
```

Listing 6.2: Initial pre-conditions as ALLOY facts

6.6 Discovered Set of Pre-conditions

```
//Discovered Model Transformation pre−condition constraints

/∗ 1. At a depth of 4 the type of an attribute has to be primitive  and cannot be a class type∗/

fact no4CyclicClassProperty{
  all a:Property | a.datatype in Class => all a1:a.datatype.attribute|a1.datatype in Class => all a2:a.datatype.
        attribute|a2.datatype in Class => all a3:a.datatype.attribute|a3.datatype in Class => all a4:a.datatype.
        attribute|a4.datatype in PrimitiveDataType
}

/∗ 2.  A Class cannot have an association and an attribute of the same name ∗/

fact noAttribAndAssocSameName{
  all c:Class, assoc:Association | all a : c.attribute | (assoc.ownedEnd == c) => a.name != assoc.name
}


/∗ 3. No cycles between non−persistent classes ∗/

fact no1CycleNonPersistent
{
      all a: Association | (a.memberEnd == a.ownedEnd) => a.memberEnd.is_persistent= True
}



fact no2CycleNonPersistent
{
      all a1: Association, a2:Association | (a1.memberEnd == a2.ownedEnd and a2.memberEnd==a1.ownedEnd) => a1.
           ownedEnd.is_persistent= True or a2.ownedEnd.is_persistent=True
}



/∗ 4.  A persistent class can't have an association to one of its general ∗/

fact assocPersistentClass
{
  all a:Association | a.ownedEnd.is_persistent=True implies a.memberEnd not in a.ownedEnd.^general
}



/∗ 5. Unique association names in a class hierarchy ∗/

fact uniqueAssocNamesInInHeritanceTree
{
     all c:Class |
    all a1:Association, a2:Association |
    (a1.ownedEnd in c and a2.ownedEnd in c.^general and a1!=a2) implies (a1.name!=a2.name)


 }


/∗ 6. A class can't be the datatype of one of its attributes (amoung all its attributes ∗/


fact classCantTypeOfAllofItsProperty
{
 all c:Class | all a: (c.attribute+c.^general.attribute) | a.datatype !=c
}

/∗ 7. A Class A which inherits from a persistent class B can't have an outgoing association with the same name
that one association of that persistent class B ∗/

fact classInheritsOutgoingNotSameNameAssoc
{
  all A:Class | all B:A.^general | B.is_persistent == True implies (no a1: Association, a2:Association |
(a1.ownedEnd = A and a2.ownedEnd=B and a1.name=a2.name))
}

/* 8. A class A which inherits from a persistent class B can't have an attribute with the same name
that one attribute of that persistent class B */


fact classInheritsOutgoingNotSameNameAttrib
{
  all A:Class | all B:A.^general | B.is_persistent == True implies (no a1: A.attribute, a2:B.attribute |
(a1.name=a2.name))
}

/* 9. No association between two classes of an inheritance tree */

fact noAssocBetweenClassInHierarchy
{
  all a : Association | all c: Class | (a.ownedEnd =c implies a.memberEnd not in c.^general) and (a.memberEnd =c
        implies a.ownedEnd not in c.^general)
}
```

Listing 6.3: Discovered pre-conditions as ALLOY facts

6.7 ALLOY Model synthesized from FSM meta-model with Facts and Partial Model Predicates

```
module metamodelFSM

open util/boolean as Bool


    sig FSM
    {
     states:set State,
     currentState: lone State,
     transitions: set Transition
    }

    sig State
    {
     label: Int,
     outgoingTransition: set Transition,
     incomingTransition: set Transition,
     fsmCurrentState: one FSM,
     fsmStates: one FSM,
     isFinal:one Bool,
     isInitial:one Bool
    }

    sig Transition
    {
     event: Int,
     target: one State,
     source: one State,
     fsmTransitions:one FSM
    }

//Meta-model constraints//

//Exactly one initial state
    fact exactlyOneInitialState
    {
     one s:State|s.isInitial == True
    }

//Atleast one final state
    fact at leastOneFinalState
    {
      some s:State | s.isFinal == True
    }

//Exactly one HFSM
    fact exactlyOneFSM
    {
     one FSM
    }


    fact sameSourceDiffTarget
    {
     all t1:Transition,t2:Transition|  (t1!=t2 and t1.source==t2.source) =>
     t1.target!=t2.target
    }

    fact setTargetAndSource
    {
     all s:State| s.incomingTransition.target = s and
     s.outgoingTransition.source=s
    }

    fact noUnreachableStates
    {
     all s: State | (s. isInitial == False) =>
     all inc1 : s.incomingTransition |
     inc1.source.isInitial = True or
     all inc2 : inc1.source.incomingTransition
     | inc2.source.isInitial = True or
     all inc3 : inc2.source.incomingTransition
     | inc3.source.isInitial = True
    }

    fact uniqueStateLabels
    {
     #State>1 => all s1:State,s2:State | s1!=s2=>s1.label != s2.label
    }

    fact containmentState
    {
     State in FSM.states
    }

    fact containmentTransition
    {
     Transition in FSM.transitions
    }

//Partial Model Facts

//Partial Model 1

    pred partialModel1_Fact
    {
     some State
    }

//Partial Model 2

    pred partialModel2_Fact
    {
     some s1:State,s2:State,t1:Transition |s1!=s2 and
     t1 in s1.outgoingTransition and t1 in
     s2.incomingTransition
    }

//Partial Model 3

    pred partialModel3_Fact
    {
     some s1:State,s2:State,s3:State,s4:State,
     t1:Transition, t2:Transition|
     s1!=s2 and s2!=s3 and s3!=s4 and s1!=s3 and
     s1!=s4 and s2!=s4 and t1!=t2 and
     t1 in s2.incomingTransition and t2 in
     s3.incomingTransition and t1 in s1.outgoingTransition
     and t2 in s1.outgoingTransition and
     s2.isInitial = True and s4.isFinal = True
    }


//Partial Model 4

    pred partialModel4_Fact
    {
     some s1:State,s2:State,s3:State,s4:State,
     t1:Transition,t2:Transition|
     s1!=s2 and s2!=s3 and s3!=s4 and s1!=s3 and
     s1!=s4 and s2!=s4 and t1!=t2 and
     t1 in s2.incomingTransition and t2 in
```

```
 s3.incomingTransition and t1 in s1.outgoingTransition
 and t1 in s1.outgoingTransition and
 s2.isInitial=True and s3.isInitial=True
}



run partialModel1_Fact for 10
run partialModel2_Fact for 10
run partialModel3_Fact for 10
run partialModel4_Fact for 10
run partialModel1_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
run partialModel2_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
run partialModel3_Fact for exactly 1 FSM,  exactly 5 State,
exactly  5 Transition, 7 int
run partialModel4_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
```

Listing 6.4: ALLOY model for FSM

List of Figures

List of Tables

Listings

Bibliography

[1] Adaptive inc, http://www.adaptive.com/.

[2] Alloy Ecore Metamodel. https://www.irisa.fr/triskell/members/sagarsen/papers/thesis/ThesisSources/alloyecore.

[3] The atl homepage, http://www.sciences.univ-nantes.fr/lina/atl/contrib/bezivin.

[4] Effective UML class diagram alloy model, https://www.irisa.fr/triskell/members/sagarsen/papers/thesis/ThesisSources/UML_CD_Small.

[5] https://www.irisa.fr/triskell/members/sagarsen/papers/thesis/ThesisSources/alloyString/attachment_download/file.

[6] http://www.irisa.fr/triskell/software-protos/Cartier.

[7] http://www.openarchitectureware.org/.

[8] Meta-model pruning kermeta implementation https://www.irisa.fr/triskell/softwares-fr/protos/metamodelpruner/.

[9] *UML 2.0 Specfication, http://www.omg.org/spec/UML/2.0/*.

[10] Xmi meta-data interchange format. http://www.omg.org/technology/documents/formal/xmi.htm.

[11] A. D. Brucker and B.Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.

[12] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, pages 44–51, November 2001.

[13] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. *Lecture Notes in Computer Science ER 2006, Springer-Verlag*, 4215:497?512, 2006.

[14] Kattepur Ajay, Sen S., and Baudry B. Pairwise interactions to effectively sample qos in dynamic web services. 2010.

[15] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmer's text editing. In *CHI 2005*, 2005.

[16] Archer J.E. Jr. and Delvin M.T. Rational's experience using Ada for very large systems. In *The First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1–B.2.5.11, Houston, Texas, NASA, June 1986.

[17] R. Bardohl, G. Taentzer, and A. Schurr M. Minas. *Handbook of Graph Grammars and Computing by Graph transformation, vII: Applications, Languages and Tools*. World Scientific, 1999.

[18] Don S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.

[19] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 2009.

[20] David Benavides, Antonio Ruiz-Cortés, don Batory, and Patrick Heymans. First international workshop on analysis of software product lines (aspl'08). In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, page 385, Washington, DC, USA, 2008. IEEE Computer Society.

[21] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz Cortés. Fama: Tooling a framework for the automated analysis of feature models. In *VaMoS*, pages 129–134, 2007.

[22] Jean Bezivin, Bernhard Rumpe, Andy Schurr, and Laurence Tratt. Model transformations in practice workshop, october 3rd 2005, part of models 2005. In *Proceedings of MoDELS*, 2005.

[23] S. Bistarelli and F. S. Santini. Soft constraints for quality aspects in service oriented architectures. In *Fourth European Young Researchers Workshop on Service Oriented Computing*, Italy, 2009.

[24] Grady Booch. *Object-oriented analysis and design with applications*. Benjamin-Cummings Publishing Co., Inc., Redmond City, CA, USA, 1994.

[25] Behzad Bordbar and Kyriakos Anastasakis. Mda and analysis of web applications. *In Trends in Enterprise Application Architecture (TEAA) in Lecture notes in Computer Science, Trondheim, Norway*, 3888:44–55, 2005.

[26] Behzad Bordbar and Kyriakos Anastasakis. Uml2alloy: A tool for lightweight modelling of discrete event systems. In Nuno Guimar?es and Pedro Isa?as, editors, *IADIS International Conference in Applied Computing*, volume 1, pages 209–216, Algarve, Portugal, 2005. IADIS Press.

[27] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of pcte and pcte+. volume 24, pages 248–257, New York, NY, USA, 1989. ACM.

[28] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[29] 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'06*, Raleigh, NC, USA, 2006.

[30] Kim B. Bruce and Joseph Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 20:50–75, 1999.

[31] M. Bruno, G. Canfora, M. Di Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In *Proc. of the 3rd Intl. Conf. in Service-Oriented Computing*, pages 87–100, Amsterdam, The Netherlands, 2005.

[32] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. The Eclipse Series. Addison Wesley Professional, 2003.

[33] Eugene C Butcher, Ellen L Berg, and Eric J.Kunel. Systems biology in drug discovery. *Nature Biotechnology*, pages 1253–1259, 2004.

[34] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for qos-aware service composition based on genetic algorithms. In *Conf. on Genetic and evolutionary computation*, pages 1069–1075, USA, 2005.

[35] J. Cardoso, J. Miller, A. Sheth, and J. Arnold. Modeling quality of service for workflows and web service processes, lsdis lab technical report pp 1-44. Technical report, University of Georgia, 2002.

[36] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[37] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, Reading, MA, USA,, 2001.

[38] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd D. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the* 15^{th} *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 130–145, 2000.

[39] David M. Cohen, Ieee Computer Society, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23:437–444, 1997.

[40] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. on Software Engineering*, 34(5):633–650, 2008.

[41] M.B. Cohen, M.B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*, pages 129–139, 2007.

[42] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, New York, NY, USA, 2006. ACM.

[43] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[44] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches, 2003.

[45] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. *Software Product Line Conference, International*, 0:23–34, 2007.

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

[47] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70?118, 2005.

[48] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism modelling and meta-modelling. In *Lecture Notes in Computer Science*, number 2306, pages 174–188, 2002.

[49] DeMillo, R. Lipton R., and F. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11(4):34 – 41, 1978.

[50] Donzeau-Gouge, V. Huet, G. Kahn, and Lang B. *Interactive Programming Environments*, chapter Programming environments based on structured editors: The Mentor experience, pages 128–140. McGraw-Hill, New York, 1984.

[51] N. Een and N. S?rensson. Minisat: A sat solver with conflict clause minimization. In *SAT*, 2005.

[52] K. Ehrig, J.M. Kuster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*, pages 156 – 170., Bologna, Italy, June 2006.

[53] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for Construction and Analysis of Systems*, Braga,Portugal, March 2007.

[54] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In *Embedded Real Time Software (ERTS)*, Toulouse, FebruaryMay 2006.

[55] Franck Fleurey, Benoit Baudry, Pierre-Alan Muller, and Yves Le Traon. Towards dependable model transformations: Qualifying input test data. *Software and Systems Modelling (Accepted)*, 2007.

[56] C. A. Floudas, A. R. Ciric, and I. E. Grossmann. Automatic synthesis of optimum heat exchanger network configurations. *AIChE Journal*, 32(2):276–290, 1986.

[57] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap,. In *FOSE '07: 2007 Future of Software Engineering*, 2007.

[58] Budinsky Frank. *Eclipse Modeling Framework*, volume 1 of *The Eclipse Series*. Addison-Wesley, 2004.

[59] Gail E. Kaiser. Incremental Dynamic Semantics for Language-Based Programming Environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, April 1989.

[60] F. Glitia, A. Etien, and C. Dumoulin. Traceability for an mde approach of embedded system conception. in ,. In *Fourth ECMDA Tracibility Workshop*, Berlin, Germany, June 2008.

[61] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating Feature Modeling with the RSEB. In *ICSR*, Washington, DC, USA, 1998.

[62] Nicolas Guelfi and Gilles Perrouin. Coherent Integration of Variability Mechanisms at the Requirements Elicitation and Analysis Levels. In Dirk Muthig and Paul Clements, editors, *Workshop on Managing Variability for SPL*, Baltimore, MD, USA, 2006.

[63] Nicolas Guelfi and Gilles Perrouin. A flexible requirements analysis approach for software product lines. In *REFSQ*, LNCS-4542, pages 78–92, Norway, 2007. Springer-Verlag.

[64] Walter J. Gutjahr. Partition testing versus random testing: the influence of uncertainty. *IEEE TSE*, 25:661–674, 1999.

[65] Habermann A.N. and David Notkin. Gandalf: Software development environments. *IEEE Trans. of Softw. Eng., SE-12*, 12:1117–1127, December 1986.

[66] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a new parallel sat solver. *Journal of Satisfiability Boolean Modeling and Computation, In special issue on Parallel SAT solving*, 6:245–262, 2009.

[67] Hans Vangheluwe and Juan de Lara. Domain-Specific Modelling with AToM3. In *The 4th OOPSLA Workshop on Domain-Specific Modeling*, Vancouver, Canada, October 2004.

[68] S. Y. Hwang, H. Wang, J. Tang, and J. Srivastava. A probabilistic approach to modeling and estimating the qos of web-services-based workflows. *Elsevier Information Sciences*, 177:5484–5503, 2007.

[69] Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *6th International Conference on Principiles of Knowledge Representation and Reasoning*, pages 636–647, 1998.

[70] J. Cabot, R. Claris?, and D. Riera. Verification of UML/OCL Class Diagrams Using Constraint Programming. In *ICST Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa'2008)*, 2008.

[71] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, April 2006.

[72] Daniel Jackson. http://alloy.mit.edu. 2008.

[73] Ivar Jacobson. *Object-oriented software engineering*. ACM Press, 1991.

[74] M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In *Proc. of the Second Intl. Conf. on Software Product Lines*, pages 15–36, London, UK, 2002.

[75] F Jouault and I Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of ACM Symposium on Applied Computing (SAC 06)*, Dijon, FRA, April 2006.

[76] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[77] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[78] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[79] Karsten Ehrig, Claudia Ermel, and Stegan Hansgen. Generation of Visual Editors as Eclipse Plug-ins. In *The 20th IEEE/ACM Internation Conference on Automated Software Engineering*, pages 134–143, 2005.

[80] Baudry B. Beneveniste A. Jard C. Kattepur Ajay, Sen S. Variability modeling and qos analysis of web services orchestrations. 2010.

[81] Stuart Kent. Model Driven Engineering. In *IFM*, pages 286–298, London, UK, 2002. Springer-Verlag.

[82] Kermeta. http://www.kermeta.org/.

[83] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, MIT, 2003.

[84] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the* 11^{th} *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer-Verlag, June 1997.

[85] J. Kienzle, N. Guelfi, and S. Mustafiz. Crisis management systems: A case study for aspect-oriented modeling, mcgill univ.

[86] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-Oriented Multi-View Modeling. In *AOSD*, pages 87 – 98. ACM Press, March 2009.

[87] Jörg Kienzle and Güven Bölükbaşi. AspectOPTIMA: An Aspect-Oriented Framework for the Generation of Transaction Middleware. Technical Report SOCS-TR-2008.4, McGill University, Montreal, Canada, December 2008.

[88] John R. Koza, Forrest H. Bennett, III, David Andre, and Martin A. Keane. Automated wywiwyg design of both the topology and component values of electrical circuits using genetic programming. In *GECCO '96: Proceedings of the First Annual Conference on Genetic Programming*, pages 123–131, Cambridge, MA, USA, 1996. MIT Press.

[89] Krysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming with ECLiPSe*. Cambridge University Press, 2007.

[90] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, 2004.

[91] Vipin Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, pages 32–44, 1992.

[92] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *MoDELS*, pages 436–450, 2007.

[93] François Lagarde, François Terrier, Charles André, and Sébastien Gérard. Extending ocl to ensure model transformations. pages 126–136. 2007.

[94] Pat Langley, Herbert A. Simon, G. Bradshaw, and y J. Zytkow. *Scientific Discovery, Computational Explorations of the Creative Mind*. MIT Press, Cambridge, Massachusetts, 1987.

[95] K. G. Larsen, U. Nyman, and A. Wasowski. Modal i/o automata for interface and product line theories. In *Joint European Conf. on Theory and Practices of Software*, pages 64–79, Braga, Portugal, 2007.

[96] Xactium Limited. Language driven development and xmf-mosaic. *Whitepaper*, 2005.

[97] Hod Lipson and Jordan B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978, 2000.

[98] Hennessy M and J.F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proc. of the 20th IEEE/ACM ASE*, NY, USA, 2005.

[99] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Int.Workshop on Description Logics (DL?2004), CEUR Workshop*, number 104, 2004.

[100] J.D. McGregor. Testing a software product line. Technical report, CMU/SEI, Technical report, 2001.

[101] Marcilio Mendoncca, Andrzej W?sowski, and Krzysztof Czarnecki. Sat-based analysis of feature models is easy. In *SPLC*, San Francisco, CA, USA, 2009.

[102] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *RE*, 0:243–253, 2007.

[103] Faucher C. Barais O. Jezequel J.M. Moha N., Sen S. Evaluation of kermeta for solving graph-based problems. *,International Journal on Software Tools for Technology Transfer (STTT), 10 p. (In Press)*, 2010.

[104] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristicphase transitions'. *Nature*, 400(6740):133–137, 1999.

[105] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving Variability into Domain Metamodels. In *MODELS*, LNCS, page 15, Denver, Colorado, USA, October 2009. ACM/IEEE, Springer.

[106] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving Variability into Domain Metamodels. In *MODELS*, LNCS, page 15, Denver, Colorado, USA, October 2009. ACM/IEEE, Springer.

[107] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06*, Bilbao, Spain, July 2006.

[108] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of MODELS/UML'2005*, LNCS, Jamaica, 2005. Springer.

[109] Clémentine Nebut, Yves Le Traon, and Jean-Marc Jézéquel. *Software Product Lines*, chapter System Testing of Product Families: from Requirements to Test Cases, pages 447–478. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.

[110] Jean Bezivin Nicolas, Nicolas Farcet, Jean marc J?z?quel, Beno?t Langlois, and Damien Pollet. Reflective model driven engineering. In *The 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003)*, pages 175–189. Springer, 2003.

[111] Niklas Een and Niklas Sorensson. An Extensible SAT-Solver. In *SAT 2003*, 2003.

[112] Niklas Een and Niklas S?rensson. MiniSat a SAT solver with conflict-clause minimization. In *SAT*, 2005.

[113] Anders Olsen, Ove Faergemand, Birger Moller-Pedersen, Rick Reed, and J.R.W. Smith. *Systems Engineering with SDL-92*. North Holland, 1995.

[114] OMG. UML 2.0 OCL 2.0 specification. Technical Report ptc/05-06-06, Object Management Group, June 2005.

[115] OMG. Mof 2.0 core specification. Technical Report formal/06-01-01, OMG, April 2006. OMG Available Specification.

[116] OMG. The uml 2.1.2 infrastructure specification. Technical Report formal/2007-11-04, OMG, April 2007. OMG Available Specification.

[117] A. Paschke and M. Bichler. Knowledge representation concepts for automated sla management. *Journal of Decision Support Systems*, 46:187–205, 2008.

[118] M. Di Penta, G. Canfora, and G. Esposito. Search-based testing of service level agreements. In *Proc. of the 9th Conf. on Genetic and evolutionary computation*, pages 1090–1097, London, England, 2007.

[119] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling automation and flexibility in product derivation. In *12th Software Product Line Conference*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[120] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. Automatic and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing*, 2010.

[121] Shari Lawrence Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, pages 219–253, 2005.

[122] M.S. Phadke. *Quality engineering using robust design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1995.

[123] Karl Popper. *The Logic of Scientific Discovery*. Hutchinson & Co., 1959.

[124] T. Massoni R. Gheyi and P. Borba. A theory for feature models in alloy. In *First Alloy Workshop*, 2006.

[125] Reiss S.P. Graphical program development with PECAN program development systems. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, Pittsburg, Pa., April 1984. ACM, New York.

[126] Isidore Rigoutsos, Aris Floratos, Laxmi Parida, Yuan Gao, and Daniel Platt. The emergence of pattern discovery techniques in computational biology. *Metabolic Engineering*, 2(3):159 – 177, 2000.

[127] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991.

[128] Sen S., Mottu J.M., and Baudry B. Automatic model generation for transformation testing. *SoSyM special issue*, 2010.

[129] S. Haar S. Rosario, A. Benveniste and C. Jard. Probabilistic qos and soft contracts for transaction-based web services orchestrations. *IEEE Trans. on Services Computing*, 1(4):187 – 200, 2008.

[130] Sagar Sen, Benoit Baudry, and Doina Precup. Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In *International Conference on the Applications of Declarative Programming*, 2007.

[131] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Domain-specific Model Editors with Model Completion. In *Multi-paradigm modelling workshop associated with MoDeLs 2007*, Nashville, TN, USA, October 2007.

[132] Sandeep Neema, Janos Szitpanovits, and Gabor Karsai. Constraint-Based Design Space Exploration and Model Synthesis. In *Proceedings of EMSOFT 2003, Lecture Notes in Computer Science*, number 2855, pages 290–305, 2003.

[133] Kathrin D. Scheidemann. Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems. In *SPLC*, pages 75–84, 2006.

[134] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE*, Minneapolis, Minnesota, USA, sept 2006.

[135] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *RE*, Minneapolis, Minnesota, USA, sept 2006.

[136] P.Y. Schobbens, P. Heymans, J.C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[137] P.Y. Schobbens, P. Heymans, J.C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[138] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select test models for model transformation testing. In *IEEE International Conference on Software Testing*, Lillehammer, Norway, April 2008.

[139] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In *ICMT*, pages 148–164, 2009.

[140] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.

[141] Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jezequel. Meta-model pruning. In *Model Driven Engineering Languages and Systems, 12th International Conference (MODELS)*, Denver, CO, USA, October 4-9 2009.

[142] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[143] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *SAT 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing*, volume 155, pages 1539–1548, June, 2001.

[144] A. Solberg, Robert France, and R. Reddy. Navigating the metamuddle. In *Proceedings of the 4th Workshop in Software Model Engineering*, Montego Bay, Jamaica, 2005.

[145] Jim Steel. *Typage de modèles*. PhD thesis, Université de Rennes 1, April 2007.

[146] Jim Steel and Jean-Marc Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.

[147] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, László Lengyel, Tihamér Levendovszky, Ulrike Prange, Dániel Varró, , and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, October 2005.

[148] Teitelbaum T. and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[149] Teitelman W. and L. Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25–34, 1981.

[150] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, 2004.

[151] Vladimir Tosic and Bernard Pagurek. On comprehensive contractual descriptions of web services. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) on e-Technology, e-Commerce and e-Service*, pages 444–449, Washington, DC, USA, 2005. IEEE Computer Society.

[152] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. Testing software product lines using incremental test generation. In *ISSRE*, pages 249–258, Washington, DC, USA, 2008. IEEE Computer Society.

[153] Tomas Vagoun. Input domain partitioning in software testing. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS) Volume 2: Decision Support and Knowledge-Based Systems*, Washington, DC, USA, 1996.

[154] A. Van Der Hoek. Design-time product line architectures for any-time variability. *Science of computer programming*, 53(3):285–304, 2004.

[155] Mandana Vaziri and Daniel Jackson. Some shortcomings of ocl, the object constraint language of uml. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, page 555, Washington, DC, USA, 2000. IEEE Computer Society.

[156] Viatra2. Department of measurement and information systems, budapest university of technology and economics. http://www.eclipse.org/gmt/VIATRA2/.

[157] Volker Haarslev and Ralf M?ller. RACER system description. In *International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 701–705. Springer, 2001.

[158] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 1–10, New York, NY, USA, 1991. ACM.

[159] Y. S. Mahajan, Z. Fu, and S. Malik. ZChaff2004: An Efficient SAT Solver. In *Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542.*, pages 360–375, 2004.

[160] I. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *ASE*, pages 409–412, Atlanta, Georgia, USA, 2007.

[161] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. on Software Engineering*, 30(5):311–327, 2004.

[162] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *Product-Family Engineering (PFE)*, volume 3014 of *LNCS*, pages 129–139, Siena, Italy, November 2003. Springer.