Adding Formal Specifications to a Legacy Code Generator

Derasari, Raj

Award date:
2021

Technische Universiteit
Eindhoven
University of Technology

Faculty of Mathematics and Computer Science

Adding Formal Specifications To A Legacy Code Generator

Raj Derasari

Supervisors:
Dr. Ivan Kurtev
Dr. Wilbert Alberts

Graduation Committee:
Dr. Ivan Kurtev
Dr. Wilbert Alberts
Dr. Natalia Sidorova

Revised: Tuesday 9th November, 2021

Dedicated to my grandparents.

Abstract

Model-driven engineering (MDE) is a development methodology that advocates the usage of models as primary software engineering artifacts. By exploiting domain knowledge, domain-specific languages (DSLs) are often developed for MDE applications, enabling development from a higher level of abstraction. In an MDE setting with DSLs, models developed in a DSL are validated and transformed to general-purpose language (GPL) programs through a code generator. Testing code generators is difficult, and is typically done manually, which increases the efforts and reduces the precision of testing.

Model-based Testing (MBT) is an approach to automate aspects of testing. It is implemented by inputting formal specifications of the software in an MBT tool, letting it generate tests, and execute each trace against the implementation via an adapter. Based on the specifications, the MBT tool predicts the output for the traces and checks for its conformance against the implementation.

The ASOME developed by Altran and ASML is a family of DSLs, accompanied by a code generator. This code generator reflects the static and dynamic semantics of the ASOME DSLs. However, the semantics have been documented and presented informally. The aim in this thesis is to explore the relevance of formal verification and MBT in the context of the ASOME code generator. For a DSL that generates into C++ code, what are the possibilities of verification with MBT? In this case, the formal semantics of the DSL are not given, so what steps need to be taken? Finally, given a set of formal specifications, can it be verified with model exploration software like Alloy, whether the intended specifications are consistent?

Considering the usage of the Alloy specification language as a formal verification approach, this thesis concludes that the DMDSL semantics can be translated into Alloy and the dynamic semantics can be verified. This thesis provides a starting point for formalizing the ASOME language specifications and verifying its dynamic semantics. It also derives that incorporating a formal verification, or an MBT-based approach, in development as early as possible (i.e. with low complexity of semantics) can enable language developers to reflect on counter-intuitive models that may be missed by manual testing.

Acknowledgments

This thesis marks the end of my Master's study at the TU Eindhoven. I am grateful for being given the opportunity to carry out my thesis project with Capgemini Engineering and ASML, on the topic of formal verification of a legacy domain-specific language.

I extend my sincerest gratitude to my supervisors Dr. Ivan Kurtev and Dr. Wilbert Alberts for their support and encouragement, guiding me throughout the project and providing an excellent collaborative environment. Thanks to Rob Caelers for the helpful demonstrations.

Thanks grandpa, for always inspiring me to be better and to pursue higher studies. Finally, I thank my parents and my sister for the consistent and unconditional support. This year was a turbulent experience for me but thanks to your support, I've made it.

Acronyms

API Application Programming Interface. In the context of this thesis, this refers to generated C++ code (functions) that can be called by the user in their code..

ASOME ASML Software Modeling Environment. Page 6.

DMDSL Domain-Interface Modeling Domain Specific Language. Page 6.

MBT Model-Based Testing. Page 3.

OOP Object-Oriented Programming. In the context of this thesis, Class and Instance/Object, Relations, Inheritance are the relevant concepts..

SAT SAT, or `SATISFIABILITY`, is an NP-complete problem. Page 34. More information is available on the Boolean Satisfiability wiki.

SUT/IUT System Under Test, or Implementation Under Test. Both terms are seen in practice, referring to the software that is being tested. This text uses the term SUT..

Contents

Preface **iii**

Contents **vi**

1 Introduction **2**
1.1 Context . . . 2
1.2 Introduction to ASOME . . . 4
1.3 Problem Statement . . . 4
1.4 Research Questions . . . 5
1.5 Thesis outline . . . 5

2 The ASML Software Modeling Environment – ASOME **6**
2.1 Introduction . . . 6
2.2 Static Semantics . . . 7
2.2.1 Repository Service Specification . . . 7
2.2.2 Domain Interfaces . . . 8
2.2.3 Entity . . . 9
2.2.4 Value objects . . . 10
2.2.5 Relations in DMDSL . . . 11
2.2.6 Repository Orientation . . . 14
2.2.7 SIRE - Separating Interface and Realization . . . 14
2.2.8 Scope Of Constructs With Example . . . 15
2.3 Dynamic Semantics . . . 16
2.4 Static Constraints . . . 18

3 Formal Specification of DMDSL Semantics **21**
3.1 Summary of assumptions . . . 21
3.2 Formal model . . . 22
3.2.1 Entity . . . 22
3.2.2 Association . . . 22
3.2.3 Multiplicity . . . 22
3.2.4 Instance . . . 23
3.2.5 Link . . . 23
3.2.6 Definition of a state . . . 23
3.2.7 Outputs for CRUD+A operations . . . 23
3.3 Constraints for well-formed model . . . 24

3.4 Auxiliary definitions and dot notation . . . 25
3.5 Repository Consistency and Desired Behavior . . . 26
3.5.1 Invariants: Repository Consistency . . . 26
3.6 Formal model: CRUD+A Operations . . . 27
3.6.1 Create Instance . . . 27
3.6.2 Add To Repository . . . 29
3.6.3 Update Instance (Association Targets) . . . 29
3.6.4 Delete Instance In Repository . . . 31
3.7 Existence of multiple initial states . . . 32
3.8 Properties of the transition system . . . 33

4 Model exploration with Alloy **34**
4.1 Requirements for a formal specification tool . . . 34
4.2 Alloy Semantics: A Primer . . . 34
4.2.1 Introduction . . . 35
4.2.2 Signature and Relation Multiplicity . . . 35
4.2.3 The small scope hypothesis . . . 35
4.2.4 Executing Alloy models . . . 36
4.2.5 Practical uses . . . 36
4.2.6 Example . . . 37
4.3 Advantages of specification with Alloy . . . 39
4.4 Disadvantages of specification with Alloy . . . 39
4.5 DMDSL Semantics with Alloy . . . 40
4.5.1 Static DMDSL Semantics and Well-formedness Constraints . . . 41
4.5.2 Definition of State in Alloy . . . 42
4.5.3 Initial State and Traces . . . 43
4.5.4 CRUD operations . . . 44
4.5.5 Repository Consistency . . . 45
4.5.6 Executing the model . . . 45
4.6 Checking for desired behavior with Alloy . . . 46
4.7 Using Alloy for Model Testing . . . 46
4.8 Conclusions . . . 47

5 Results **49**
5.1 Models without an initial state . . . 49
5.1.1 Cyclicity of Associations . . . 49
5.1.2 Association Targets . . . 50
5.1.3 Non-causal associations . . . 50
5.1.4 Miscellaneous Counterexamples . . . 51
5.2 Verifying dynamic semantics (Valid Initial State) . . . 52
5.2.1 CRUD Traces: Correctness . . . 52
5.2.2 Traces without the Update operation . . . 52
5.2.3 Deadlock scenarios in Alloy with bounded verification . . . 54
5.3 Conclusions and insights . . . 55

6 Discussion **56**
6.1 Related Work . . . 56

6.2 Future Directions 57

7 Conclusions **58**
7.1 Answers to the research questions 59

Bibliography **61**

A DMDSL specification in Alloy **64**
A.1 Create 64
A.2 Update 65
A.3 Delete 65

Chapter 1

Introduction

1.1 Context

Modern software systems are complex and based on large codebases. Such complex systems are not easy to verify for correctness. Thus, abstracting on some aspects of development can be beneficial and reduce the person-hours required in developing and maintaining systems. To this extent, model-based techniques are a suitable tool facilitating software development from a higher level of abstraction, allowing reduced complexity for developers.

In *model-driven engineering* (MDE), a model is the fundamental unit of structuring information [14, 19]. Model-driven techniques are based on creating and exploiting domain models for exchanging information. *Domain-Specific Languages* (DSLs) are often used in MDE to define domain models in a simpler syntax [3]. This allows engineers having domain experience to express designs in familiar languages. If the DSL can be parsed and compiled, then code generation and verification is possible. DSLs are often integrated in language workbenches in development environments (IDE) such as Eclipse [11].

Usually, models developed in a DSL are validated and transformed to a *General-Purpose Language* (GPL), e.g. C++ or Java in order to execute them. This is done by developing a *code generator*, which is a typical *exogenous* model transformation [16] – the same model contents are expressed in a different language. Thus, a DSL user only needs to know the DSL semantics, which is at a higher abstraction than the generated code. Additionally, models may also be created via a graphical editor (within the IDE). In general, a DSL will always have *static* semantics, and it may also have *dynamic* semantics. The static semantics include the definition and intention of a model in the DSL, while the dynamic semantics maps the DSL model to a corresponding execution ('runtime') behavior [23].

The reliability of a DSL for defining valid models depends on the correctness of the code generator. Flaws in the model transformation can lead to uncompilable generated code; alternatively, logical errors in syntactically correct generated code can lead to unforeseen runtime inconsistencies. Typically, correctness of a code generator is verified manually: either by inspecting the model transformation (M2M, M2T [5]); or by providing input models and either inspecting generated code or running test cases on the built version of the generated code. Inherently, manual testing leads to lower precision and requires more patches later. The possibility of improving testing so that it is more reliable, can later allow focusing on new features

as there are no bugs to be fixed.

Model-based testing (MBT) is a black-box testing approach that advocates for a formal model of the software alongside the implementation, to provide higher precision in testing. The term 'specifications' is often used in place of formal models. In the specification, the *intended* behavior of the software are defined. These are compared against the implementation, referred to as the Implementation/System Under Test (IUT/SUT) in practice. There are two methods of testing the IUT - Online, and Offline. In the offline method, all tests are generated first, then the aggregated test suite is translated, and executed on the SUT. In the online testing approach, test generation and execution is simultaneous, but an adaptor is required to translate and execute test cases on the SUT.

The MBT setup in the online testing context involves three components, as shown in Figure 1.1. These are the formal model m, the tester suite, and the adaptor a. The formal model m defines the formal specifications of the intended definition of the implementation. Then, a tester component parses m and generates test cases for it, as a sequence of actions ('traces') in m, starting from a valid initial state. Finally, an adapter a needs to be defined, which converts a test case to a format recognized by the implementation and executes it against the IUT. In the case of a DSL, this adapter generates the relevant GPL (e.g. C++) file. Note that, as a consequence, MBT in the case of DSLs also requires that the DSL must have dynamic semantics.

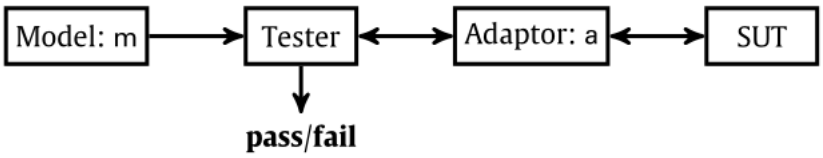

Figure 1.1: Model-based testing. (Ref. Figure 1 in [21])

In the model m, the formal specifications must be defined based on a relevant paradigm. There are multiple paradigms for describing formal specifications [25]. The paradigms of transition-based specifications and state-based specifications are commonly seen in model-checking and model-based testing software. Transition-based specifications are suited for **reactive** systems [7], that use *labelled transition systems* (LTS) [24] or statecharts [13] to define the formal specifications. In essence, transition-based specifications are suitable when the IUT behavior is based on input actions and system reactions, for example a vending machine. Alternatively, state-based specifications [25] define the contents of a state as the formal specification, for instance in the form of a sets or ordered sequences. In this case, transitions between different states require changes to the states' contents. An example would be a file system with CRUD (Create, Retrieve, Update, Delete) operations on files and folders, where a state is defined by the set of contents of the file system. While there are overlaps in these paradigms, the emphasis is either on the SUT system contents ('properties of system'), or on the SUT reaction ('what the system does'). (For a precise comparison of state-based and transition-based paradigms, refer to [18, 22]). For testing a DSL and code generator, it may be possible to use either state-based or transition-based formal specifications – this depends on the DSL semantics and its runtime behavior.

1.2 Introduction to ASOME

The ASML SOftware Modeling Environment ASOME, was developed as a software following the ASML DCA Pattern (separation of Data, Control, Algorithms). The DCA pattern is based on the assumption that algorithms, control, and data can better be verified in isolation and then combined. This is based on the validity of tools like ASD[1] and CoCo[2]. By using ASOME, engineers can define the data and behavior in separate environments. ASOME has been in development since 2015, and due to its practical usage, the implementation is developed but the formal specifications are not defined.

The "Domain Interface Modeling DSL" (DMDSL) is a part of the ASOME family of DSLs, to handle the Data aspect of the DCA pattern. Other DSLs exist for modeling Control, Algorithms, and Systems, but they are not considered in this project. Here, the aim is to discover the suitability of formal verification and MBT for ASOME, for a subset of its semantics, and check the correctness of the code generator.

In the DMDSL, a user defines models that represent information in constructs like **entities** and **ValueObjects**, across **domain interfaces**. According to the dynamic semantics, entities can be instantiated and these **instances** can be stored to a repository and the repository content is manipulated (retrieved, updated, deleted) at runtime. These are the CRUD+A operations on entities and instances. With an Object-oriented programming (OOP) analogy, entities are akin to classes, and instances at runtime are akin to class objects. The concept of **multiplicity** is used to introduce bounds on the number of instances. Additionally, it is possible to create **relations** between some constructs: these are binary relations with a domain and a range, but the relations may also have properties. Static constraints are defined with the declarative language OCL[3]. These constraints are used to enforce validity in DMDSL, so that inconsistent models cannot be compiled into generated code.

Usually, when GPLs like C++ or Java are mentioned, the *runtime* is expected to mean 'after executing the bytecode'. In the context of DMDSL, *runtime* refers to actions and functions written in the generated C++ files, which will later be compiled to bytecode and executed. Briefly, the dynamic semantics of ASOME allow for operations such as **creating, adding, retrieving, updating**, and **deleting** of instances in the repository. These are referred as the **CRUD+A** operations. It is important to ensure that given a consistent behavior of the repository, executing any of these operations still leads to consistent behavior. Therefore, the precise definition of these operations in the form of a formal specification is necessary. The DMDSL is explained in detail in Chapter 2.

1.3 Problem Statement

The static and dynamic semantics of the DMDSL are given informally in text and diagrams. Usage of natural language for semantics discussion can cause misconceptions between language engineers, which can lead to gaps in the implementation. This allows creating inconsistent models that follow the defined static semantics of DMDSL, but lead to unexpected runtime behavior. As the code generator in ASOME was tested manually, this suggests that some static constraints on the model might be missed, or that the operations in the dynamic semantics

[1] https://verum.com/asd/
[2] https://www.cocosimulator.org/
[3] Chapter 7, The OCL Specification

might need refinement. Consequently, it is possible to define inconsistent models. (The definition of inconsistency is addressed later.)

Due to lack of the formal specifications of the DSL, a crucial step in this project is to understand and formally specify the semantics. Due to time constraints and its inherent complexity, the complete formal specification of ASOME cannot be documented. Only the subset of the static semantics with relevant dynamic semantics for addressing repository consistency is applied. The primary goal is to consider the applicability of formal verification for the DMDSL, which is a functional real-world industrial use case since 2016. Formal verification is applied by specifying the DMDSL semantics in the formal specification tool **Alloy**. By defining Boolean assertions for consistency, it is possible to explore the DMDSL specifications and discover models that violate the assertions. Then, either the formal specifications or the implementation must be addressed to fix the context of the violation.

For the formal verification of the code generator with a tool, an appropriate MBT solution must be selected. In this tool, the formal specifications can be modeled, and traces can be generated. The requirements from a formal verification tool, for testing the DMDSL spec are documented. In this thesis, Alloy[4] is used to verify the specification of DMDSL. Considering the maturity of ASOME and how its correctness (and complexity) have gradually increased, the benefits and disadvantages of using Alloy on an industrial scale software can be discovered. Considering the problem statement and approach, the following research questions are noted.

1.4 Research Questions

RQ1. How should the semantics of DMDSL be precisely specified, considering the informally defined semantics?

RQ2. What is a suitable tool for formal verification of DMDSL semantics, and what is required?

RQ3. What are the benefits and disadvantages of a formal verification approach in a real-world industrial context like DMDSL?

1.5 Thesis outline

The structure of this thesis is as follows: Chapter 2 introduces the ASOME software and DMDSL, with its static and dynamic semantics. Chapter 3 describes the formal specifications of ASOME for the considered subset of semantics. Chapter 4 introduces Alloy as a specification language and the AlloyAnalyzer as a supporting tool in Section 4.2 and the DMDSL specifications in Alloy are provided in Section 4.5. In Chapter 5, the results of the experiments in Alloy are presented and validated against the C++ implementation. Chapter 6 discusses related literature and also potential future tracks for the project. Chapter 7 concludes this thesis.

[4] https://alloytools.org/

Chapter 2

The ASML Software Modeling Environment – ASOME

This chapter aims to introduce the reader to the static and dynamic semantics of the DMDSL in ASOME. The version of ASOME used in this project was ASOME v3.6.0.20210316-1551, and is approximately 5 years into development. An introduction to ASOME is given in Section 2.1. The static semantics used to define a model in ASOME are given in Section 2.2. The dynamic semantics of DMDSL allow the user to execute operations on the generated code, and this is explained in Section 2.3.

2.1 Introduction

ASOME is developed in the Eclipse IDE and includes *static* and *dynamic* semantics (also referred to as *runtime semantics*) for the consisting DSLs. For creating models, textual and graphical representations are provided. ASOME includes a family of DSLs developed in Altran and ASML. These reflect the Data-Control-Algorithms (DCA) architecture pattern adopted by ASML. This text only concerns the DMDSL. The Domain Interface Modeling DSL (DMDSL) is used to define data models.

While a detailed explanation of the DMDSL semantics will follow in the next sections, a brief introduction is given here, from a top-to-bottom approach. The first construct in a DMDSL model is a Repository. This repository has a Service Specification and a Service Realization (ref. SIRE - Section 2.2.7). Even for the most basic models, a default realization is always generated. The repository specification service may specify at least 1 domain interface. A Domain Interface can then specify a data model (Entities, ValueObjects, attributes, primitive types, enumerations, and constants) and their relations (associations, specializations, and compositions). The relations are as such: associations are unidirectional from a source entity to a target entity, and an association multiplicity for both the source and target; specialization is a unidirectional relation to define hierarchies and inheritance among entities; and Composition is a relation that expresses the containment of a target ValueObject to a source Entity or ValueObject.

If the defined repository service specification is valid then its code can be generated. In this generated code, entities can be instantiated and added to a repository, which is a storage

mechanism for entities. The CRUD operations can be done to manipulate the repository, i.e. the instances in the repository. This is analogous to classes and instances in object-oriented programming. After its creation, an entity instance is still not significant until it is added and stored in a repository, after which other operations can be done on it. Among all the data objects, the generated code includes identifiers for entity instances. Contrarily, value objects do not have identifiers and are not stored in a repository, they are only contained by entities. In the generated code, CRUD operations are used on entity instances, and their associations. The entity multiplicity limits the number of instances of the entity that can be created. The association multiplicities (having the same interpretation as in UML) require a number of instances to participate in the association, so that the association relation is satisfied. The entity and association multiplicities must be respected at runtime otherwise an exception is returned. Note that if the minimum multiplicity for entity is non-zero, then instances of this entity must be provided during construction of the repositories that contain them (ref. Constructor Delegates, Sec 2.2.3).

Static constraints are used to enforce validity of a model, otherwise the runtime operations can become inconsistent or erroneous. A model is valid if it violates no static constraints. To enforce static constraints, the Object Constraint Language (OCL) [26] is used. OCL is a declarative language, and is used for defining rules on Meta-Object Facility (MOF)[1] metamodels, including UML. The Eclipse Modeling Framework (EMF) also provides an MOF, on which the OCL constraints are applied. This way, DMDSL developers can prevent the users from creating invalid models. The correctness of the OCL constraints significantly affects the correctness of the DMDSL. Constraints are declarative and can be as simple as ensuring that the maximum multiplicity is greater than the minimum multiplicity, but also complex recursive calls such as an anti-cyclicity constraint that would need to check for closed paths of varying length for binary (or higher order) relations.

If all OCL static constraints are satisfied by the model, then it may have the code generated for it. This is known as a valid model w.r.t. the DMDSL semantics. A code generator compiles the DMDSL model into C++ code, after which it is said to be in *runtime*, at which point the dynamic semantics (runtime semantics) apply. It may be possible that a model is valid for DMDSL, but it may never get initialized with dynamic semantics - concluding that maybe more constraints are required to reject such a model during validation.

Section 2.2 explains the static constructs in the DMDSL, and Section 2.4 describes the constraints to ensure validity of models.

2.2 Static Semantics

2.2.1 Repository Service Specification

A repository service specification ('RS') is the first element created in a DMDSL model. An RS defines all the repositories in a DMDSL model. There may be multiple RS in a model. Each RS has a port, say p, that is used to connect to all Domain Interfaces provided on that port. Note that an RS is only the specification of the contents. The Repository Realization (RR) handles implementation-relevant functionalities (dynamic semantics and code generation, ref. Section 2.2.7). The RR enables variants of an interface to be realized by one repository realization and allows for model evolution. An example of a specification (RS) is shown below. In

[1] www.omg.org/mof

this example, the 'p1' port defines the domain interface `iDomain1DM` on the repository spec `sService1RS`, and it has an implicit default repository realization, not visible on the diagram.

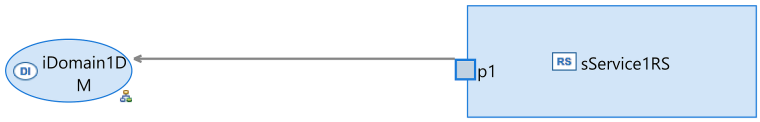

Figure 2.1: Repository Service Specification (RS)

2.2.2 Domain Interfaces

A repository service may provide or require domain interfaces via ports. A domain interface (DI) in the DMDSL is where the basic static constructs are defined. In a DI, it is possible to define entities, associations, value objects, etc. There can be multiple DI in a model and they may be provided from the same port. The purpose of a domain interface is to enable model evolution and separation of clients: An entity with some properties in a domain interface can exist (with the same or different name) as an entity in another domain interface, with different properties. This involves repository realizations, which is not in the scope of this text.

This text considers the context of a single domain interface, which means **each entity is distinct and has one representation**. A domain interface can have a number of constructs defined in it. The graphical editor ensures that all properties are visible onscreen with text and icons as shown in Figure 2.2. Notice that different arrows indicate what type of relation is defined. `Entity1` and `Entity2` have an association relation. `Entity2` and `Entity3` have a specialization relation (`Entity3` is the parent entity of `Entity2`, i.e. `Entity2` inherits from `Entity2`, with an OOP analogy). The value objects are contained within `Entity1` by a composition relation.

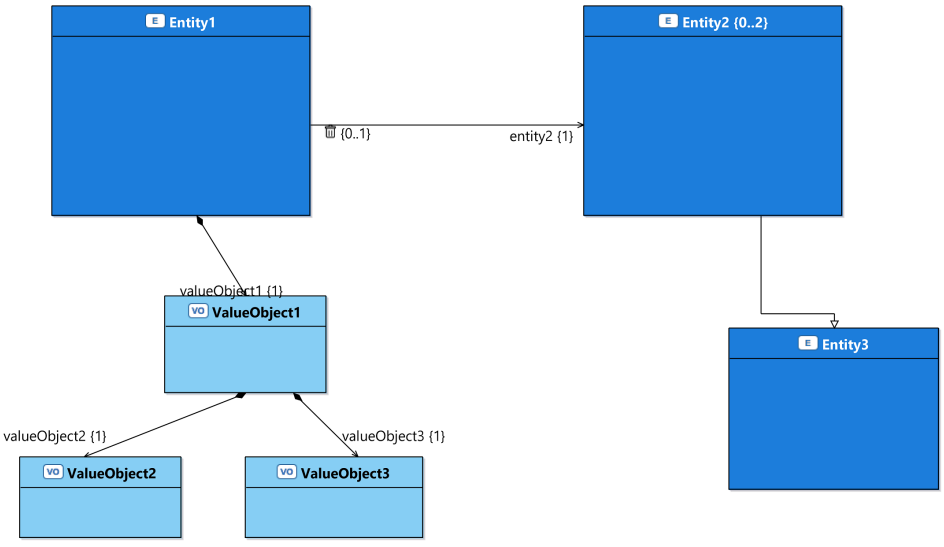

Figure 2.2: Example: contents within a domain interface.

2.2.3 Entity

An entity in DMDSL is used to store data via attributes, constants, and ValueObjects. In the UML analogy it is similar to a Class. It might be possible to instantiate an entity at runtime (depending on its properties). Each instance of the entity is identified by a unique identifier (two entity instances may be instantiated with the same properties and contents, but they are distinct). An entity has the following properties:

1. Constructability: It takes values from { Constructable, Unconstructable }. An entity with the **Constructable** property can be instantiated from the domain interface it is defined in. However, if the Constructability property for an entity is **Unconstructable**,then the entity cannot be instantiated **via the interface**, although depending on the repository realization (Sec 2.2.7) it might be instantiated from another interface.

2. Mutability: It takes values from { Editable, Uneditable, Immutable }. If the Entity has Mutability = **Editable** in an interface, then its instances can be updated at runtime by a user accessing the instance via the interface. If Mutability = **Uneditable** in an interface, entity instances cannot be updated by a user accessing it from this interface (however other interfaces that provide the entity with the property as Editable, can still be used to modify the instances). If Mutability = **Immutable**, instances of the entity cannot be updated by anyone once they are stored in the repository. The static constraints o on the repository realization enforce that the same entity cannot be simultaneously defined as Editable and Immutable.

3. Deletability: It takes values from { Deleteable, Undeletable, Undestructable }. If the Entity has property Deleteability = **Deleteable**, then its instances can be deleted from the repository via the interface. If the property Deleteability = **Undeleteable** in an interface, then its instances cannot be deleted via this interface (although users of another interface where the property is Deleteable can delete an instance). However, if Deleteability = **Undestructable**, then instances of the entity can **never** be deleted once stored in the repository.

While the entity properties allow operations on them, the entity may have its **entity multiplicity** defined as well. The entity multiplicity defines a lower bound (LB/minimum) and an upper bound (UB/maximum). The multiplicity constrains how many instances of the entity may exist at a given time, and it must be verified that the dynamic semantics respsect this.

An example of an entity definition is given below, in graphical form:

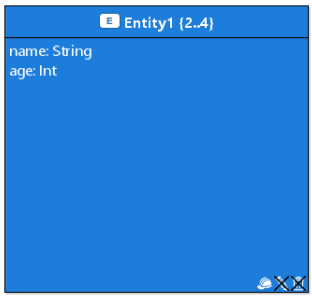

Figure 2.3: An entity with attributes and properties

The properties of the entity (and other DMDSL constructs) can be seen in the IDE:

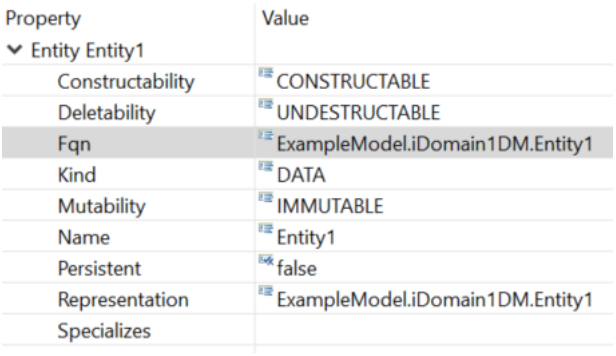

Figure 2.4: Properties of the Entity construct

Repositories

Once created, instances of an entity may be stored in the entity repository. A repository is created for each Entity type in the model. The term *repository* in general is used to refer to combination of all the repositories of all entities. Entity instances can still exist outside their repositories, thus the multiplicity is considered as such: For creation of an instance, the maximum multiplicity is respected (regardless of repository). However, the minimum multiplicity is always validated for instances in the repository. The association multiplicity is always considered for instances in the repositories.

Constructor Delegates

The intended semantics is that at first, the repositories are empty. However, in case of a minimum multiplicity that is non-zero, the repositories have to be populated by using constructor delegates. Observe the multiplicity (2..4) of `Entity1` in Figure 2.3. In such cases, *before runtime*, instances of this entity are expected to be populated in the repository, such that the entity and association multiplicities are valid before runtime. A constructor delegate allows creating instances and storing to repositories, and while this is still done by a user via a software interface in the generated code, it is considered to be *before* runtime. When the repository is initialized, it can be expected that the constructor delegates have provided entity instances in their corresponding repositories so that all the entity and association multiplicities are satisfied.

Note that, constructor delegates are essentially C++ APIs, like the other generated code. These delegates are generated only for entities having a non-zero minimum multiplicity, but in the delegates the user can still create instances of any entity, regardless of its minimum multiplicity.

2.2.4 Value objects

Unlike entities, Value objects are in essence tuples of attributes, without an identifier. Consequently, they are also not stored in a repository. However, ValueObjects can be instantiated and contained by other constructs (i.e. Entities or other ValueObjects), and they inherit the mutability property from their containing entity (Composition relation, ref Sec 2.2.5).
Two ValueObjects may be instantiated with the same values but then they are not distinguishable (in contrast to entities).

An example model with ValueObject is given below. In this setup, `ValueObject2` and `ValueObject3` contain some integer representations of components of a velocity vector, and then `ValueObject1` may be some algorithmic evaluation of its corresponding attributes, e.g. $netVelocity = \sqrt{verticalVelocity^2 + horizontalVelocity^2}$, received from `ValueObject2` and `ValueObject3`.

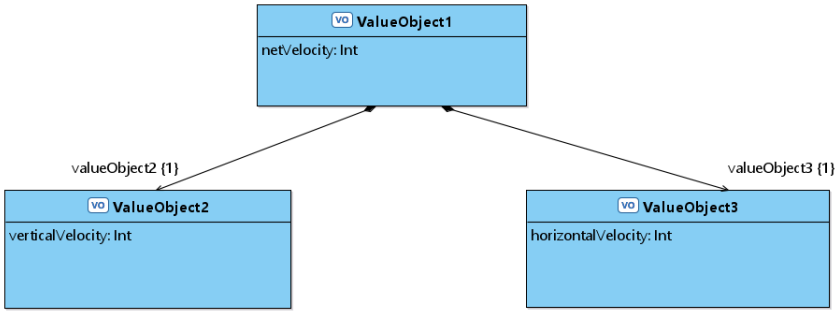

Figure 2.5: The ValueObject construct

2.2.5 Relations in DMDSL

There are three relations in DMDSL, and they are all binary relations. These are Association, Specialization, and Composition. The relation significant to this text is Association, as the complexity of the relations given the limited time for the assignment makes it tough to include all of these. The relations are described below.

Association

An association is a binary relation from an Entity to an Entity. It is unidirectional (source → target). The semantics are similar to a UML association relation, except that an Entity cannot be associated to itself (non-reflexive). An association also has properties: **Multiplicity** and **Cascade Delete**. In the generated code, associations are addressed from the source entity instance.

- At runtime, an association is instantiated/populated when a link from an instance (of the source type) is created, to an instance of the target type. These links between instances contribute to the source and target multiplicities of the association. The multiplicity of an association is defined both at source and target ends, and it indicates the number of instances that can participate in the association, linked to a given instance.

- The Cascade Delete property is defined for the purpose of *explicit lifecycle modeling*. The Cascade Delete is theoretically defined for source and target ends of the association, however the considered version of ASOME (v3.6.0) only considers the source cascade delete (SCD). The SCD property indicates that, given an association that connects a source instance to a target instance, if the target instance is deleted then whether the source instance should also be deleted. The link between the source and target is deleted regardless of the SCD property. But if the SCD property is true, the source instance is also deleted. The formal specification in this text also considers Target Cascade Delete (TCD) – the semantics for handling the deletion of a target instance if a source instance in an association (with TCD enabled) is deleted.

An example of an association is given below. Observe the Target multiplicity (1..3) - this suggests that an instance of the `Entity1` type cannot exist in the repository, without at least w1 instance of the `Entity2` type in the repository, as the association is violated otherwise. Additionally, the trashcan icon at the source end of the association indicates that SCD property for the association is enabled. So, if an instance Y of `Entity2` is deleted, then all links of `associationExample` where Y was a target, will have the *source instance* deleted as well.

Figure 2.6: Association relation: multiplicity and cascade

Referring to the model in Figure 2.6, the following is required for the association to be valid in the definition of a consistent repository:

- For an instance x of `Entity1` in the repository, there must be at least 1, and upto 3 links to some instance(s) of `Entity2` for `associationExample`. Additionally, all these instances must be in the repository.
- For an instance y of `Entity2` in the repository, there can be at most 2 instances of `Entity1` for `associationExample` in the repository that link to y.

Navigating over an association, from a source instance, returns an unordered collection of instances that allows duplicates, and this is known as the *Bag semantics*. In the implementation, *ordered* associations are also possible, pointing to a *Sequence* of instances. In both cases, duplicates are allowed, i.e. the same target instance may occur multiple times as a target of an association, and the association is considered valid. If the navigation of an association returned a *Set* of instances, then duplicates would not be allowed. An example is given below in Figure 2.7.

Figure 2.7 presents the comparison of bag semantics and set semantics for association validity. Here, the minimum association target multiplicity is 2. In the Bag and Sequence semantics (as the implementation provides), it is possible to create `Instance2` and refer to it twice, pointing from `Instance1`. This satisfies the minimum target multiplicity for `Instance1`. Alternatively, in the set semantics, this would not be possible and another instance of `Entity2` would be required.

Specialization

A specialization relation is defined to introduce classification hierarchies among entities. It is a binary relation where the domain and range is always an Entity, which must be within the same interface. Now, a *child* entity cannot have more than one *parent* entity. The parent is called the *specialized* entity and the child is called the *specializing* entity. Any specialized entity is abstract in the implementation and cannot be instantiated. All the properties, at-

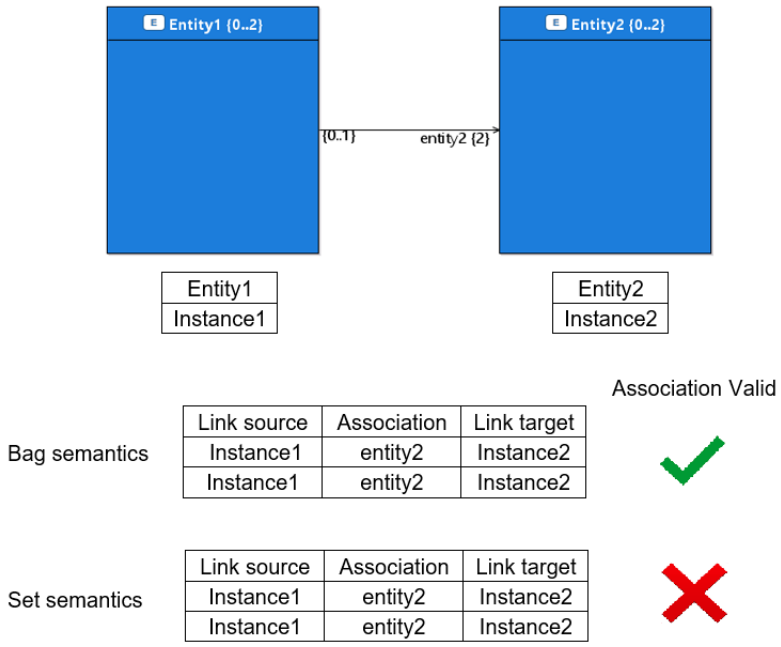

Figure 2.7: Bag semantics against set semantics

tributes, associations are inherit by the specializing entity. The specialization relation has no properties like multiplicity but it does introduce constraints on the multiplicities of participating entities: The minimum and maximum multiplicity of a parent entity, must be the sum of all its specializing entities.

An example of the specialization relation is given below. Notice the association `assoc0` that is inherited in `Entity1_Child`. The upper bound of `Entity1`, 5, is the sum of all the child entities = 5.

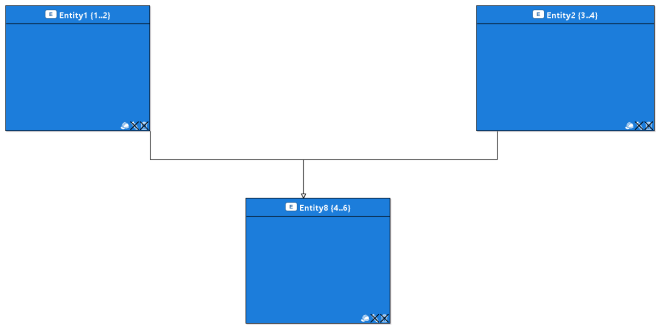

Figure 2.8: Specialization relation

Composition

A composition relation defines the containment relation of a ValueObject to an (Entity or ValueObject), i.e. the target of a composition is a ValueObject. The composition from an Entity to a VO essentially introduces the VO as an attribute of the containing Entity.

Similarly, a ValueObject can contain another object. Thus, the target of a composition is always a ValueObject. An example of composition has been given above in Figure 2.5.

2.2.6 Repository Orientation

In the DMDSL, repositories can be defined in two orientations: Reference orientation and Clone orientation. When requested for an entity, a *reference-oriented* repository provides a reference to the contents. The updating of contents is implicit and instantaneous, which improves efficiency but can lead to interference between clients, moreover, rolling back is complicated. In a *clone-oriented* repository, a clone of the instance is produced and returned, this means that updates must be saved explicitly - This allows data to be prepared before committing, rolling back, and keeping clients isolated. The orientation is set up for each entity type in the implementation.

Given the available time and complexity of the semantics, all repositories are assumed to provide reference oriented entities. This simplifies the retrieval of an entity since clones do not have to be addressed.

2.2.7 SIRE - Separating Interface and Realization

As this work considers only the default/generated realization, this section is not relevant for the formal specification.
A repository specification service may provide more than one domain interface. For example, consider a case where one client may create or update entities from one interface, and another client may only read the entities from another interfaces. Through the repository realization (RR), a domain interface may expose only a part of the realization to the client. The RR is defined for a given specification (RS), and provides a realization for all elements in all provided interfaces. The realization contains all the elements of the domain interface. An RR realizes ValueObjects, Entities, relations. However, a realization element may realize more than one interface element, as evidenced in Figure 2.9.

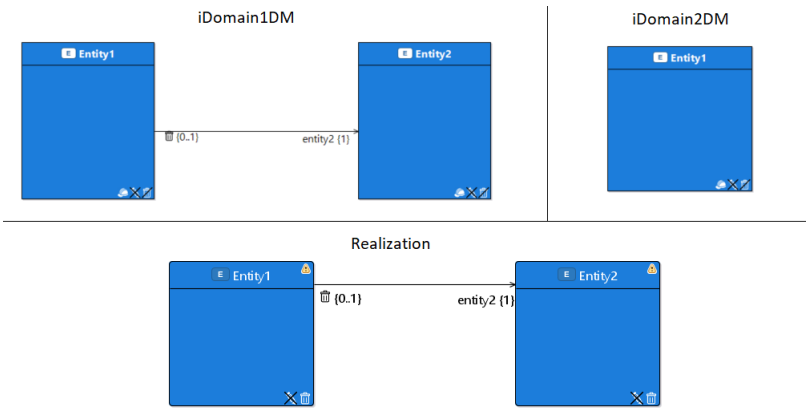

Figure 2.9: Repository realization properties

In `iDomain1DM`, there are two entities and an association. In `iDomain2DM`, there is one entity, which is also the source of the association in `iDomain1DM`. The realization specifies the properties for all the entities and the association discovered in the specification. Now, `Entity1` in `iDomain2DM` may be allowed to be Deleteable and the model would still be valid. However, a

user with access to `iDomain1DM` cannot delete instances of `Entity1` as for them, `Entity1` is Undeleteable.

In the Realization given above, `Entity1` realizes `iDomain1DM.Entity1` and `iDomain2DM.Entity1`.

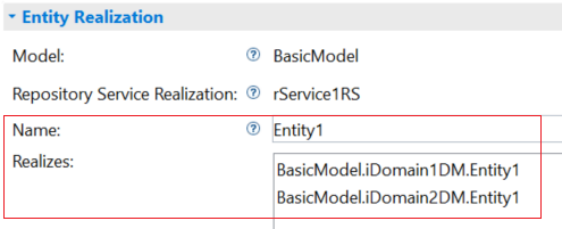

Figure 2.10: The properties of realization

In the RR semantics, there is no Constructability property for Entity. The multiplicity must have the same values in the interface and realization. The realization entity Mutability property can only be Editable or Immutable, and realization entity Deleteability property can only be Deleteable or Undestructable. Therefore, the Mutability property as Editable or Uneditable in an interface, maps to Editable in the realization; and Immutable in an interface maps to Immutable in the realization. Similarly, the Deleteability property as Deleteable or Undeleteable in an interface maps to Deleteable in the realization, and Undestructable in interface maps to Undestructable for realization. The association relation also has to be realized - If an association has Cascade Delete enabled in the interface, it must have Cascade Delete enabled in the realization. The aforementioned semantics are enforced with static constraints.

A repository specification with only one provided interface can always have a default realization automatically generated for code generation. A default realization provides a 1-1 mapping between interface and realization. For more than one provided interface, a default realization is possible but might need to be modified.

2.2.8 Scope Of Constructs With Example

In this thesis, the scope is limited to within a Domain Interface, including Entities and Associations with respect to all their properties and multiplicities. The specialization relation and repository realizations are not addressed. A simple valid DMDSL model based on the included semantics is given in graphical and textual form. This model will be used as a running example.

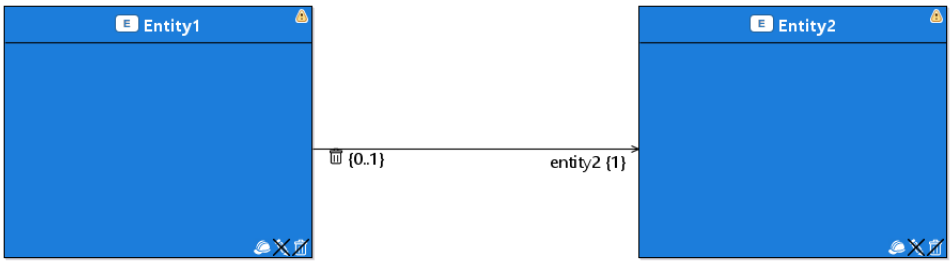

Figure 2.11: Sample model of DMDSL features

```
Model BasicModel {
    RepositoryService sService1RS {
        Provided DataPort p1 {
            interfaces : iDomain1DM
        }
    }
    DomainInterface iDomain1DM {
        Entity Entity1 [0, inf] {
            lifecycle : Constructable Immutable Undeletable
            associations :
                [0, 1] entity2 : Entity2 [1, 1] unordered {
                    lifecycle :
                        on source delete : target stays
                        on target delete : source dies
                };
        }
        Entity Entity2 [0, inf] {
            lifecycle : Constructable Immutable Undeletable
        }
    }
}
```

2.3 Dynamic Semantics

The code for a valid ASOME model can be generated with an Implementation Model - a collection of *recipes*. This text only considers the default implementation recipes. The figure below shows the model transformation of the code generator.

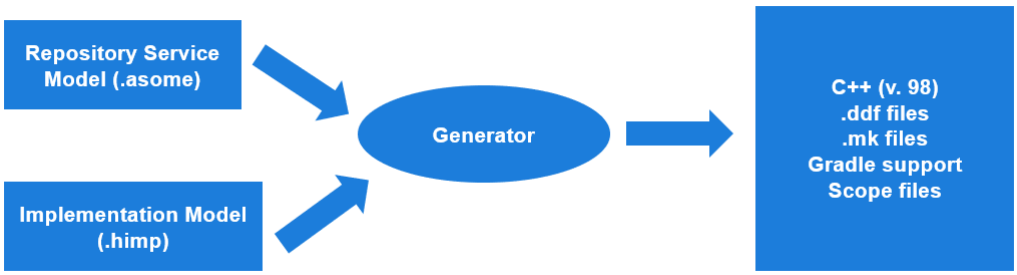

Figure 2.12: ASOME code generation transformation

The dynamic (runtime) semantics are applicable to the generated code, defining the logic that is possible on the code by making C++ function calls. In these semantics, entities can be created, added, retrieved, updated, and deleted, i.e. the CRUD+A operations. Following are important constraints about the dynamic semantics:

- Entities can be added to repository only one at a time. There is no support for transaction-like semantics (where multiple objects added to repository simultaneously)
- At any moment, any CRUD+A operation could be executed. It is expected that the CRUD+A operations should respect the consistency of the repository. Briefly, the definition of repository consistency is the satisfied multiplicity of entities and associations.
- To preserve multiplicity of associations, keeping in mind that only one entity can be

added to repository at a time, the minimum multiplicity of association source is forced to be 0. (Though target multiplicity minimum can be non-zero). This way, a target can always be added to repository first, then the source can be added, and consistency is preserved.

Creation of Entity Instance

To create an entity, a reference to its factory class is required. Within a domain interface, this is possible only if the Entity type is Constructable. For any outgoing associations that this instance participates in, the association targets are also required, and satisfying the multiplicity of this association is a precondition. The number of instances of this entity should not exceed its maximum multiplicity - otherwise, a runtime exception error occurs.

Consider the `BasicModel` that is presented on Page 15. From the `iDomain1DM` interface, to create an instance of `Entity1`, the following logic is required. Since `Entity1` is a source of an association, its instance cannot be created unless an instance of `Entity2` exists. Note that this is a simple example, using only a read-only repository.

```
1  SB = ServiceBundle();           // an sb is always required
2
3  E2Factory = SB -> getEntity2Factory();      // get the factory
4  Instance_E2 = E2Factory -> create();        // create an instance
5  E2_Repository = SB -> getForEntity2();             // get repository access
6  E2_Repository -> add(Instance_E2 -> getID());   // Instance_E2 is added now
7
8
9  E1Factory = SB -> getEntity1Factory();             // gets the factory
10 Instance_E1 = E1Factory -> create(Instance_E2 -> getID());  // 1 instance of
       entity2 required
11 E1_Repository = SB -> getForEntity1(Instance_E1);    // get repository access
       for Entity1
12 E1_Repository -> add(Instance_E1 -> getID());     // store instance in
       repository
```

Listing 2.1: DMDSL runtime semantics

Adding Instance to Repository

The addition to repository is shown in Listing 2.1 with `E2_Repository`. When an instance is added to the repository, its associations have to be checked for (1) source multiplicity validity (2) target multiplicity validity, and (3) all association targets of this instance must in their repository. The violation of any of these conditions causes a runtime error.

Updating an Instance

An entity instance can be always be updated via any interface as long as it is not stored in the repository. If it is stored in the repository, then it can be updated *iff* it has the Mutability = Editable. There are two possible updates on an entity instance – *attribute* updates, and *association* updates. Attribute updates are analogous to using get() and set() methods in OOP to update class objects. These are not in the scope of the text. Association updates are important: Using the update method, all the target instances for an instance can be set, for a given association. In essence, for a given instance, only one association is allowed to update in the update operation, and the new collection of targets for the association is provided, overwriting the older association targets. With the update operation, validating the association source and target multiplicities is important.

Deleting an Instance from repository

An instance may be deleted if it is in the repository and the entity type has the Deleteable property in the interface. An instance can still be deleted if the property is Undeleteable in the interface, if it has a cascade deletion that is triggered outside the interface. However, if the entity has the property as Undestructable, then it can never be deleted from the repository. The following is observed in the deletion of an instance:

- The instance itself is deleted. The intended semantics is that the instance is removed from the repository. The implementation differs from the specifications about this - in the implementation, the instance is removed from the repository but still exists in memory. Therefore, all the associations of the instance are preserved but they are not counted in validating the association. In the formal specification, the deleted instance and its links do cease to exist.
- If the instance is a target of an association link, then the source instance will be deleted if the association has the Source cascade Delete (SCD) property enabled. Otherwise, the source instance will be updated: its references to the target instance will be removed.
- Since the association source multiplicity is forced to have a minimum of 0, deletion never violates the source multiplicity. However, it might violate target multiplicity, if the static constraints and deletion function are not correctly specified. Thus, the cascaded deletion semantics are important.
  Cascade delete: The cascaded deletion of entity instances following the associations introduces complexity in the deletion operation. Consider that the source cascade delete (SCD) is enabled, then it must be verified that deleting an instance of the target entity does not violate the multiplicity of the source entity. This is ensured by applying static constraints, as in Section 2.4. If SCD and TCD are both enabled for an association, then it is necessary to guarantee that the navigation over the association does not cause a loop and always terminates.

2.4 Static Constraints

These constraints are define to exclude models that can cause inconsistencies at runtime. In the DMDSL, there are many static constraints based on the semantics of realizations and multiple interfaces. The considered subset of DMDSL semantics is within a single interface. For this subset, the relevant static constraints are listed below in an informal language.

- Multiplicities: The minimum multiplicity must be non-negative, and the maximum must be positive. The maximum must be greater than or equal to the minimum.
- Entities: In the Entity multiplicity, if the minimum is equal to the maximum, then the entity must be Undestructable.
- Associations:
  - The association source multiplicity must always have a minimum of 0.
    As only one instance can be added to repository at a time, this constraint is necessary to allow maintaining repository consistency – With this constraint, the target instance may be added to the repository first, while zero source instances point to it. No such constraint is applied on the target, however. Therefore, a source in-

stance is in the repository may require > 0 target instances to be in the repository. By the definition of association, and considering that only instance is added a time, either the source or target minimum multiplicity must always be 0, otherwise the consistency of associations is violated for non-zero minimum multiplicities). The implementation enforces this on the source. Figure 2.13 shows an invalidated model as the source minimum is set to be non-zero.

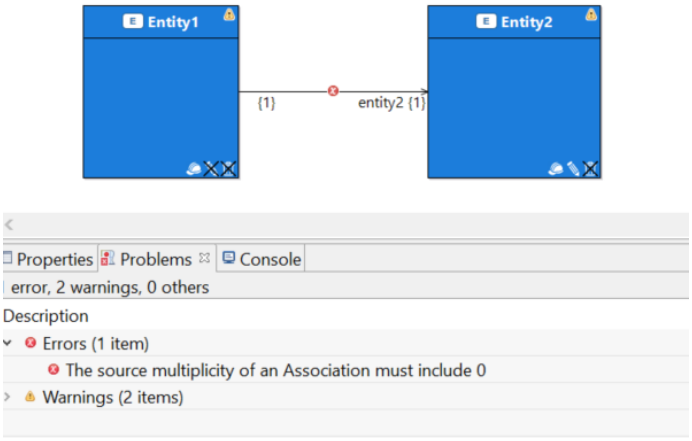

Figure 2.13: Invalid: Minimum association source multiplicity must be 0

– If the target entity of an association can be deleted, then, either the SCD should be enabled, or the source entity must be made Editable. If SCD has been enabled, then the source Entity must also have the Deleteable property. Additionally, the source association multiplicity always has a minimum of 0, as the previous constraint defines.

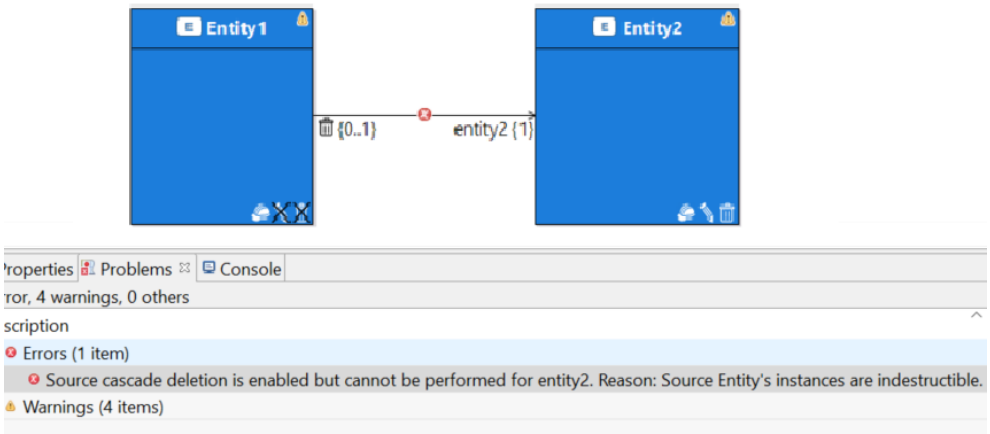

Figure 2.14: Invalid: SCD is enabled but the source entity is Undestructable

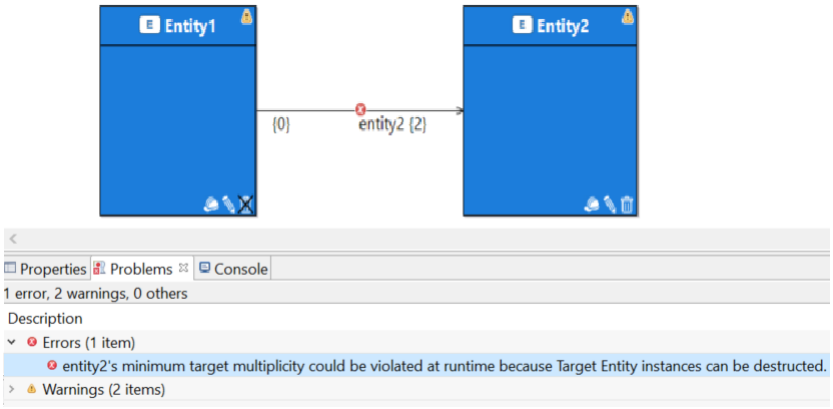

Figure 2.15: Invalid: Target instances can be deleted but target multiplicity = 2, expected 0

– If the source entity of an association can be deleted, then the TCD may be made enabled, which enforces the target entity to be Deleteable. Alternatively, if the TCD is disabled, there are no concerns with static constraints. Since the source instance is deleted, the association is vacuously satisfied: the source minimum is 0 and can never be violated, and the target minimum may be non-zero but as the source instance ceases to exist, target multiplicity has no premise.

Chapter 3

Formal Specification of DMDSL Semantics

This chapter introduces the formal specification of DMDSL. Considering the complexity of the DMDSL, only a portion of the formal specification is documented in this thesis. The main goal is to check the correctness of the CRUD+A operations in terms of repository consistency. The approach to define the specification is to first specify the contents within an interface (here, it is the entities and associations) and the related CRUD+A operations. In this chapter, Section 3.2 introduces the syntax and sets required for representing the DMDSL semantics. Then, to formalize the dynamic semantics, the notion of State and contents of a state are introduced. Finally, the specification for the CRUD+A operations is given in Section 3.6.

3.1 Summary of assumptions

A summary of the assumptions of the semantics is given below:

- Only the default repository realization generated by ASOME is considered. This is possible when there is only one domain interface. The formal model here does not address repository realizations and multiple domain interfaces. (Introducing customized repository realizations would be more convenient with the top-down approach where the specifications for multiple domain interfaces can be defined first.)
- Bag semantics: The ASOME implementation uses Bag and Sequence semantics. Therefore, duplicates are allowed and elements may be ordered (Sequence) or unordered (Bag). For a given instance, navigating over the outgoing links of a particular association returns a Bag. A bag is an unordered sequence of elements and allows duplicates. The text assumes the unordered associations.
- Repository orientation: It is assumed that all entities are stored with reference-oriented repositories. This simplifies the Retrieve() operation, as the instance is not duplicated, updates are immediate, and the state of the repository does not change.
- The generalization/specialization relation is not addressed.

3.2 Formal model

The first step here is to define the structure of an ASOME model. The formal model of the DMDSL is presented as the model M: M is a tuple < Entity, Association >. Here, Entity is the set of the entity types, and the Association is the set of the associations in the model.

3.2.1 Entity

Let *Entity* represent the set of entities. The definition of an element $e \in Entity$ is a tuple:

$$e =< C, M, D, N >$$

This element $e \in Entity$ is a tuple of properties: Constructability (C), Mutability (M), Deleteability (D), and Multiplicity (N).
Constructability (C) = True means the Entity is Constructable.
Constructability (C) = False means the Entity is Unconstructable.
Mutability (M) = True means the Entity is Editable.
Mutability (M) = False means the Entity is Uneditable or Immutable.
Deleteability (D) = True means the Entity is Deleteable.
Deleteability (D) = False means the Entity is Undeleteable or Undestructable.
Multiplicity (N) = <minimum, maximum> is the entity multiplicity.

3.2.2 Association

First, a property of the association ends (source/target) is required. This is,

$$\texttt{AssociationEndProperty} = < \texttt{Cascade}, \texttt{Multiplicity} >$$

`Cascade` is a Boolean and represents the cascaded deletion property of that association end. The `Multiplicity` is a multiplicity pair (min, max).

Now, let the set Association define the associations in DMDSL. The definition of an Association element $a \in Association$ is:

$$a = < \texttt{source}, \texttt{target}, \texttt{SProperty}, \texttt{TProperty} >$$

Here, $a.source \in Entity$ and $a.target \in Entity$. Furthermore, $SProperty \in$ AssociationEndProperty, and $TProperty \in$ AssociationEndProperty. For reference, `SProperty.Cascade` represents the Source Cascade Delete (SCD), and `TProperty.Multiplicity` is the association target multiplicity.

3.2.3 Multiplicity

A multiplicity N is a pair (minimum, maximum), such that
Multiplicity $= \{(min, max) \mid min >= 0 \wedge max >= min \wedge max > 0\}$

Formal model: Syntax for dynamic semantics

For verification of the DMDSL, the semantics of the model M will be defined as a transition system. A state in this system is the contents of the repositories and the instances not yet in the repositories, and this state must also indicate links between the instances. The transitions correspond to the CRUD+A operations that can be executed on the repository. The following definitions assume a fixed model M (as defined in 3.2): the entities and associations in M do not change once instantiated, and for the model M there is a transition system corresponding to it that the specification defines.

3.2.4 Instance

In the dynamic semantics, an entity can be instantiated. Let the set $Instance$ be the universe of all instances that can exist. Each instance x has an entity type, that it was created of. Thus, *type* is a property of an instance. Let $type : Instance \rightarrow Entity$ be a function on an instance x such that $type(x) = y$ where $y \in Entity$ is the entity of instance x. This function is populated with the creation of an instance, and deleting an instance

3.2.5 Link

A *link* is an instance of an association. The $Link$ is a multiset[1] that captures all links at runtime. A link $l \in Link$ is a tuple:
$l =< \texttt{source}, \texttt{association}, \texttt{target} >$: Instance × Association × Instance. The well-formedness of a link is given in Equation (3.9), so that an element in $Link$ corresponds to valid instances and associations.

Corresponding to the *bag semantics* that represent the intended specifications, there may be two links p and q such that $p.s = q.s, p.a = q.a, p.t = q.t$.

3.2.6 Definition of a state

A *State* S is defined as the tuple $S =< I, REPO, L, type, output >$. Here, $S.I \subseteq Instance$ is the set of instances in the state S. Then, $S.REPO$ is the set of instances that are stored in a repository in state S. For an instance to be in the repository, it has to be defined as an instance first, thus it must be in $S.I$. Thus, it follows that $S.REPO \subseteq S.I$. Finally, $S.L \subseteq Link$ is the set of links in the state S. Therefore, in a given state S, the created instances are in $S.I$, the added instances are in $S.REPO$, and the association links are in $S.L$. The *type* function maps the instances in state S to their corresponding entities in model M, and is updated with the Create and Delete operations. The *output* is used to present the success or error from the operation in the previous state. It takes values from an enumeration that defines all possible outputs in the transition system, to represent the cases where a CRUD+A operation fails.

The transition corresponding to a successful CRUD+A operation will modify the contents of a state, going into a distinct new state. In this text, the Create, Update, Add, Delete are the relevant operations. Create will modify $S.I$ and $S.L$; Add will modify $S.REPO$; Update will modify $S.L$; Delete will modify $S.I, S.REPO, S.L$. Section 3.6 explains this in detail.

3.2.7 Outputs for CRUD+A operations

In the transition system for model M, from a state s, one of the CRUD+A operations can make a transition to the next state s' if the operation is successful. Alternatively, the operation may fail due to a variety of possible errors, which would not change the contents of the state. The purpose of the *output* property of a state is to indicate the status of the CRUD operation. A transition indicating a failed CRUD+A operation has a non-successful output, should have the target as s itself. **A failure case does not mean that the transition cannot be taken.** The transition is still possible, but the result state is different.

The outputs can be represented with the following labels as an enumeration:

- `Entity_MultiplicityMaximum`: If the maximum multiplicity of an entity is violated, its instances cannot be created.

[1] A multiset is a modification to the definition of set, that allows for multiple instances of the set elements

- `Entity_MultiplicityMinimum`: If the minimum multiplicity of an entity is violated, its instances cannot be deleted.
- `Entity_Unconstructable`: Attempt to create an entity that is unconstructable
- `Entity_Immutable`: Attempt to modify/update an entity that is immutable
- `Entity_Undestructable`: Attempt to delete an entity instance whereas its type is undestructable
- `Entity_UnexpectedAssociation`: Attempt to create links for an instance, for some association where the entity is not a source.
- `Entity_MissingAssociation`: Attempt to create an entity instance where a required association is not provided.
- `Association_SourceMaximum`: The maximum source multiplicity of the association is violated (during create or update)
- `Association_TargetMaximum`: The maximum target multiplicity of the association is violated (during create or update)
- `Association_TargetMinimum`: The minimum target multiplicity of the association is violated (during update or delete)
- `Link_TargetNotInRepository`: For a link, the source instance is found in the repository but the target instance is not.
- `Instance_NotInRepository`: Attempt to Update or Delete an instance not in the repository.
- `Instance_AlreadyInRepository`: Attempt to add an instance to repository, after it was already added.

3.3 Constraints for well-formed model

To create a specification corresponding to the intended semantics, the following constraints are enforced on the formal model.

1. **Entity**: The constraints in Section 2.4 are formalized to:

$$\forall e \in \texttt{Entity} \mid e.N.minimum = e.N.maximum \implies E.Deleteability = False \quad (3.1)$$

$$\forall e \in \texttt{Entity} \mid e.Deleteability = True \implies e.N.minimum = 0 \quad (3.2)$$

   Note: The implementation does not enforce a minimum multiplicity of 0 for a deleteable entity. This is enforced only when an entity is a source entity in an association with SCD enabled. In this case, the constraint would instead be:

$$\forall e \in \texttt{Association.source} \mid a.SProperty.Cascade = True \implies e.N.minimum = 0 \quad (3.3)$$

2. **Acyclic model:** Informally, this constraint requires that associations do not lead to cyclicity, i.e. a source entity of an association is not reachable by following the association targets. Given $e \in Entity$,

Let $targets(e) = \{a.target \mid a \in \texttt{Association} \wedge a.source = e\}$.
Let $\hat{}$ denote the transitive closure operator on the binary relation $targets$.

$$ACYCLIC :: \forall e \in Entity, e \notin \ ^{\wedge}targets(e) \tag{3.4}$$

3. **Association Source Multiplicity**: The source multiplicity minimum is always 0.

$$\forall a \in Association, \texttt{a.SProperty.multiplicity.minimum} = 0 \tag{3.5}$$

4. **Association Target Properties**: For associations, if the target entity is deleteable then the source entity should be cascade deleted, unless the association allows a source instance to refer to 0 targets:

$$\forall a \in \text{Association}, \texttt{a.target.Deleteability} = True \implies (\texttt{a.SProperty.cascade} = True \vee \texttt{a.TProperty.multiplicity.minimum} = 0) \tag{3.6}$$

5. **Association Cascade Properties**: For associations, the source or target cascade delete being enabled requires that the corresponding entity is deleteable and forces a minimum entity multiplicity of 0. If the cascade is disabled, then the source instance must be editable (mutable).

$$\forall a \in Association, \texttt{a.SProperty.Cascade} = True \implies (\texttt{a.source.multiplicity.minimum} = 0 \wedge \texttt{a.source.Deleteability} = True) \tag{3.7}$$

$$\forall a \in Association, \texttt{a.SProperty.Cascade} = False \implies \texttt{a.source.Mutability} = True \tag{3.8}$$

6. **Link validity**: For a link to be valid, the source and target must be of the corresponding types from the association.

$$\forall p \in Link, type(p.s) = p.a.source \wedge type(p.t) = p.a.target \tag{3.9}$$

3.4 Auxiliary definitions and dot notation

Given: S: a state, and a model $M =< Entity, Association >$ where $E \in M.Entity$: and $A \in M.Association$, and $Z =< s', a', t' >$: a set of links where $Z.s' \in Instance, Z.a' \in Association, Z.t' \in Instance$; and Z is assumed to be well-formed. these definitions will be frequently used.

Equation 3.1 finds all outgoing associations from an entity type. Equations 3.2 and 3.3 are to find the number of instances in state S having entity type E. Equations 3.4 and 3.5 find the instances in state S, for the source and target of an association, respectively. Equation 3.5 finds the outgoing links from a given source instance s', in a set Z. Equation 3.6 finds the incoming links to a target instance t'. Equations 3.5 and 3.6 are on an arbitrary set of links Z as this set may be all the links in a state, or a set of links to delete.

$$OUT(E) = \{a \in \text{Association} \mid \text{a.source} = E\} \tag{3.10}$$
$$I_E(S) = \{i \in S.I \mid type(i) = E\} \tag{3.11}$$
$$R_E(S) = \{i \in S.REPO \mid type(i) = E\} \tag{3.12}$$
$$R_{source}(S, A) = \{i \in S.REPO \mid type(i) = \texttt{A.source}\} \tag{3.13}$$
$$R_{target}(S, A) = \{i \in S.REPO \mid type(i) = \texttt{A.target}\} \tag{3.14}$$
$$L_{OUT}(Z, s', a') = \{p \in Z \mid p.s = s' \wedge p.a = a'\} \text{ (outgoing in set Z)} \tag{3.15}$$
$$L_{IN}(Z, a', t') = \{p \in Z \mid p.a = a' \wedge p.t = t'\} \text{ (incoming in set Z)} \tag{3.16}$$

Dot notation: It is useful to have the dot notation for extracting all elements of a property in the set. For example, the *Link* set has elements such as $l =< s, a, t >$ where s, t are instances and a is an association. Therefore, *Link.a* is used to extract all associations in the set, and is of type *Association*. In set logic this is equivalent to:

$$Link.a = \{p.a \mid p \in Link\}$$

3.5 Repository Consistency and Desired Behavior

The contents of the state S, that is $S.I, S.REPO, S.L$ are important in this specification, and the dynamic semantics (CRUD+A operations) must ensure that the consistency of the contents is maintained in all states. The repository consistency criteria are represented as *invariants*, that must be respected in every state of the transition system. In essence, **the CRUD+A operations should not violate the consistency of the repository**. If a violation is observed, then the CRUD+A operations must be refined, or the model should be invalidated by adding static constraints. The *well-formedness* properties are defined to assure that over time, desired behavior is observed in the system. Here, the well-formedness checks that at least one operation on the entity for its enabled properties can be executed, e.g. if an entity is Constructable, it should have an instance created at least once, and if it is Deleteable, it should have an instance deleted at least once.

3.5.1 Invariants: Repository Consistency

Let $|X|$ be the representation of the cardinality of set X.

Entity Multiplicity

Given entity E, let $min =$ `E.multiplicity.minimum` and $max =$ `E.multiplicity.maximum`. Then,

$$\forall S \in State, min \leq |R_E(S)| \wedge |I_E(S)| \leq max \tag{3.17}$$

Informally, this expresses that the number of created entities is within the minimum and maximum for the entity defined in model M, for every state in the transition system.

Association Multiplicity

The association multiplicity is the conjunction of the source multiplicity and target multiplicity, which can be independently evaluated.
Source Multiplicity:

Given an association A,
Let max = `A.SProperty.multiplicity.maximum`

$$\forall S \in State, y \in R_{target}(S, A) \mid |L_{IN}(S.L, A, y)| \leq max \tag{3.18}$$

Informally, the source association multiplicity is validated by looking at each existing instance of the target entity type, and counting the number of source instances that point to it. This number must be less than the maximum source multiplicity. As the source minimum is always 0, the lower bound is always satisfied and need not be checked.

Target Multiplicity:
Given state S and association A,
Let min = `A.TProperty.multiplicity.minimum` and
max = `A.TProperty.multiplicity.maximum`.

$$\forall x \in R_{source}(S, A) \mid min \leq |L_{OUT}(S.L, x, A)| \leq max \tag{3.19}$$

Informally, the target association multiplicity is validated by looking at each instance of the source entity type i.e. $R_{source}(S, A)$), and counting the number of target instances that it points to (L_{OUT}). This number must be greater than (or equal to) min and less than or equal to max. Here, the target minimum multiplicity may be non-zero so the minimum must be validated.

Link Validity

A link l is valid in state S *iff* both the source and target instances are in $S.I$. Furthermore, if $l.s \in S.REPO$ then $l.t \in S.REPO$

$$\forall S \in State,\ l \in S.L \mid l.s \in S.I \wedge l.t \in S.I \tag{3.20}$$
$$\forall S \in State,\ l \in S.L \mid l.s \in S.REPO \implies l.t \in S.REPO \tag{3.21}$$

3.6 Formal model: CRUD+A Operations

The CRUD+A operations in the DMDSL specification define the transition from the current state to the next state. Each CRUD+A operation refers to a transition. The specifications of the CRUD+A operations now follow.

3.6.1 Create Instance

Let s be a state in the transition system. Given $e \in$ `Entity`, the entity to instantiate, $links \subseteq Link$, for the target instances corresponding to associations of e

Let $new \in I$ indicate the instance that is created by this operation. The output for Create, is instance new such that $type(new) = e$ and the set $links$ is added to the $s.L$.

Hence, the operation in s has the signature $Create(s, e, links)$. The next state s' is derived as follows.

Preconditions

The following preconditions must be met for successful creation

1. e.Constructability = True

2. $|I_e(s)| < e.\texttt{multiplicity.maximum}$
3. $type(new) = e \wedge new \notin s.I$
4. $OUT(e) = \{l.a \mid l \in links\}$ (all associations should be in targets)
5. $\forall l : links \mid l.s = new \wedge l.t \in S.I$
6. Target multiplicity:

$$\forall a \in OUT(e) \mid \texttt{a.TProperty.multiplicity.min} \leq |L_{OUT}(s.L, new, a)| \leq \texttt{a.TProperty.multiplicity.max}$$

7. Source multiplicity:

$$\forall a \in OUT(e),\ y \in I_{a.target}(s),\ |L_{IN}(s.L, a, y) + L_{IN}(links, a, y)| \leq \texttt{A.SProperty.multiplicity.maximum}$$

Relation to the next state

The relation between the next state s' and the current state s in the creation:
$s'.I = s.I \cup new$
$s'.L = s.L \cup links$
$s'.REPO = s.REPO$
$s'.output = Success$
$type[new \rightarrow e]$: The *type* function is updated for instance *new*. Let $type' : Instance \rightarrow Entity$, such that $type'(i) = type(i),\ \forall i \in s.I$ *and* $type'(new) = e$

Failure cases

Entity instance creation can fail due to a variety of reasons. The goal here is to consider all possible cases where the creation can fail. The failure cases can be determined by applying the negation of each precondition. In the failure cases, the other contents of the new state retain the same contents, i.e. $s'.I = s.I \wedge s'.REPO = s.REPO \wedge s'.L = S.L$.

1. $\texttt{e.Constructability}$ = False $\implies s'.output = \texttt{Entity\_Unconstructable}$
2. $|I_(e)(S)| = e.multiplicity.maximum \implies s'.output = \texttt{Entity\_MultiplicityMaximum}$
3. $links.a \nsubseteq OUT(e) \implies s'.output = \texttt{Entity\_UnexpectedAssociation}$
4. $OUT[e] \nsubseteq links.a \implies s'.output = \texttt{Entity\_MissingAssociation}$
5. $\exists a \in OUT(e), |L_{OUT}(links, new, a)| > \texttt{a.TProperty.multiplicity.maximum}$
   $\implies s.output = \texttt{Association\_TargetMaximum}$
6. $\exists a \in OUT(e), |L_{OUT}(links, new, a)| < \texttt{a.TProperty.multiplicity.minimum}$
   $\implies s.output = \texttt{Association\_TargetMinimum}$
7. Source maximum multiplicity violation:

$$\exists a \in OUT(e), x \in R_{a.target}(s) \mid \mid L_{IN}(S.L, a, x) \cup L_{IN}(links, a, x) \mid > \texttt{A.SProperty.multiplicity.maximum} \implies s'.output = \texttt{Association\_SourceMaximum}$$

3.6.2 Add To Repository

Let s be a state in the transition system, and s' is the next state. Let $x \in \texttt{s.I}$ be the instance to add to repository. Hence, the operation in s has the signature $Update(s, s', x)$.

Preconditions

The following preconditions must be met for successfully adding a created instance to repository:

1. $x \in s.I$
2. $x \notin S.REPO$
3. $\forall l : s.L, l.s = x \implies l.t \in s.REPO$

Relation to the next state

The actions in addition to repository are:
$s'.I = s.I$
$s'.L = s.L$
$s'.REPO = s.REPO \cup x$
$s'.output = Success$

Failure cases

Adding an instance to repository can fail if the association multiplicities for it are not already satisfied. The source multiplicity is expected to not be violated since it is a precondition to the instance creation. It may also be that a user tries to add an already added instance to repository. These cases are handled here. In the failure cases, the other contents of the new state retain the same contents, i.e. $s'.I = s.I \wedge s'.REPO = s.REPO \wedge s'.L = S.L$.

1. $x \in s.REPO \implies s'.output = \texttt{Instance\_AlreadyInRepository}$
2. $\exists p : s.L \mid p.s = x \wedge p.t \notin s.REPO \implies s'.output = \texttt{Link\_TargetNotInRepository}$

3.6.3 Update Instance (Association Targets)

Let s be a state in the transition system, and s' is the next state. Let $x \in s.I$ be the instance to update. In the association update, an instance can receive a new set of association targets, for a single association. Thus, a required input is the association targets, $links$, corresponding to the instance new. Hence, the operation in s has the signature $Create(s, s', x, links)$.

Preconditions

The following preconditions must be satisfied for successfully updating an association of an instance:

1. $type(x)$.Mutability = True
2. $x \in s.I$
3. $|links.a| = 1 \wedge links.a \in OUT(x.type)$
4. $\forall l : links, l.s = x \wedge l.t \in S.REPO$
5. Target multiplicity: For the instance to update, count the number of provided targets

in the parameter.

$\forall a \in links.a$, `a.TProperty.multiplicity.min` $\leq |links| \leq$ `a.TProperty.multiplicity.max`

6. Source multiplicity: For all target instances, count the number of existing links pointing to it and the number of times it occurs in the parameter *links*.

$\forall a \in links.a,\ y \in I_{a.target}(s),$
$$|L_{IN}(s.L, a, y) + L_{IN}(links, a, y)| \leq \texttt{A.SProperty.multiplicity.maximum}$$

Relation to the next state

The actions in updating an instance are (note the set difference $-$ as all old links outgoing from x are removed):
$s'.I = s.I$
$s'.REPO = s.REPO$
$s'.L = s.L - L_{OUT}(s, x, links.a) \cup links$
$s'.output = Success$

Failure cases

Updating an instance in the repository can fail for multiple reasons: Trying to update an instance of an immutable entity type, or invalid association source/target multiplicities, or a case where the updating instance is in the repository but the association target is not. These cases are handled here. In the failure cases, the other contents of the new state retain the same contents, i.e. $s'.I = s.I \wedge s'.REPO = s.REPO \wedge s'.L = S.L$.

1. $type(x)$.Mutability $= false \implies s'.output =$ `Entity_UpdateImmutableType`
2. $\exists l : links | l.s = x \wedge l.t \notin s.REPO \implies s'.output =$ `Link_TargetNotInRepository`
3. Target minimum multiplicity violation:

$$|links| < links.a.TProperty.multiplicity.minimum$$
$$\implies s'.output = \texttt{Association\_TargetMinimum}$$

4. Target maximum multiplicity violation:

$$|links| > links.a.TProperty.multiplicity.maximum$$
$$\implies s'.output = \texttt{Association\_TargetMaximum}$$

5. Source maximum multiplicity violation:

$$\exists a \in OUT(e), x \in I_{a.target}(s) \mid$$
$$|L_{IN}(S.L, a, x) \cup L_{IN}(links, a, x)| > \texttt{A.SProperty.multiplicity.maximum}$$
$$\implies s'.output = \texttt{Association\_SourceMaximum}$$

3.6.4 Delete Instance In Repository

Let s be a state in the transition system, and s' is the next state. Let $x \in s.REPO$ be the instance to delete. The transitive closure operator $\hat{}$ is again useful, to navigate over all the links, looking for cascade deletion candidates. Due to the cascade delete, some auxiliary relations will be useful to indicate the elements involved in deletion. The $connectsSCD$ relation maps an instance y to all the instances connected to it, via any association with SCD enabled. In this case y is always a target of the link. Similarly the $connectsTCD$ relation maps instance x to all instances connected to x via an association with TCD enabled, wherever x is the source of a link.

$$connectsSCD(y, S) = \{l.s \ for \ l \in S.L \mid l.t = y \wedge \texttt{l.a.SProperty.Cascade} = True\} \tag{3.22}$$

$$connectsTCD(x, S) = \{l \ for \ \ \in S.L \mid l.s = x \wedge \texttt{l.a.TProperty.Cascade} = True\} \tag{3.23}$$

$$markedForDeletion(x, S) = \{x \cup \ ^\wedge connectsSCD(x) \cup \ ^\wedge connectsTCD(x)\} \tag{3.24}$$

$$\begin{aligned} deleteLinks(x, S) = \{&l \in S.L \mid p.s \in markedForDeletion(x, S) \\ &\vee \ p.t \in markedForDeletion(x, S)\} \end{aligned} \tag{3.25}$$

Preconditions

The following preconditions must be met for successfully updating an association of an instance:

1. $type(x)$.Deleteability = True
2. $x \in s.REPO$
3. $R_{x.type}(s) > \texttt{x.type.multiplicity.minimum}$
4. $\forall y : markedForDeletion(x, S), \texttt{y.type.Deleteability} = True$
5. Entity multiplicity of remaining entities:
   $\forall e \in \texttt{Entity}, |R_e(S) - markedForDeletion(x, S)| >= e.multiplicity.minimum$
6. Target multiplicity of remaining links/associations:

$$\begin{aligned} &\forall a \in \texttt{Association}, p \in \{R_{a.source}(S) - markedForDeletion(x, S)\}, \\ &\quad |L_{OUT}(S.L - deleteLinks(x, S), p, a)| \geq \texttt{a.TProperty.multiplicity.minimum} \end{aligned} \tag{3.26}$$

Relation to the next state

The preconditions and static constraints are expected to respect the repository consistencies, and appropriately delete instances connected by the association cascade delete property (source and target). The actions are:
$s'.I = s.I - markedForDeletion(x, S)$
$s'.REPO = s.REPO - markedForDeletion(x, S)$
$s'.L = s.L - deleteLinks(x, S)$
$s'.output = Success$

The *type* function is updated to exclude all deleted instances: Let $type' : Instance \rightarrow Entity$ such that $type'(i) = type(i), \forall i \in s'.I$. In s', let $type = type'$.

Failure cases

The static constraints are defined such that the deletion operation does not violate the multiplicity of an association. As mentioned in Equation (3.3), any entity that participates in an association with the cascade enabled, is required to have a minimum multiplicity of 0. Therefore, failure of deletion can happen only if entity multiplicity of the entity $type(x)$ is violated. In the failure cases, the other contents of the new state retain the same contents, i.e. $s'.I = s.I \wedge s'.REPO = s.REPO \wedge s'.L = S.L$. Finally, the deletion can fail in the following cases:

1. $x \notin s.REPO \implies s'.output =$ `Instance_NotInRepository`
2. $\exists y \in markedForDeletion(x, S) \mid type(y).\text{deleteability} = false \implies s'.output =$ `Entity_DeleteOnUndestructable`
3. Let $e = type(x)$, then

$$|R_e(s) - \{y \in markedForDeletion(x) \mid type(y) = e\}| < e.multiplicity.minimum \implies s'.output = \texttt{Entity\_MultiplicityMinimum}$$

3.7 Existence of multiple initial states

A DMDSL model is a valid model if it follows all static constraints, and there must always a transition system corresponding to a valid DMDSL model. To define a transition system, an initial state (or a set of initial states) is required.

A basic initial state for the transition system is one where the contents of the sets I, $REPO$, and L are empty. This case is possible only if all the entities in M have a minimum multiplicity of 0. However, the converse is not true: if all the entities in the model have a minimum multiplicity of 0, it is still possible to populate the contents of the initial state, as long as the maximum multiplicity is not violated. Therefore, the initial state may have ≥ 0 instances.

For the bag semantics and the static constraints of DMDSL, and the logical fact that any number of instances (below the maximum multiplicity) can be created before runtime, *an initial state for the a valid model M in the DMDSL specification, will always exist.* In this text, a valid initial state of the transition system for a model M is specifying constraints for an initial state in terms of entity and association multiplicities.

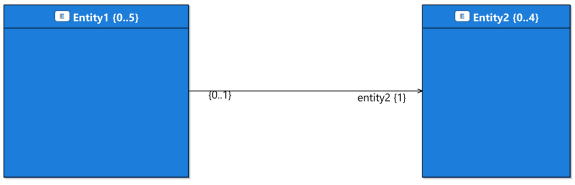

Figure 3.1: An example model in DMDSL

Consider Figure 3.1 - there are multiple initial states for this model. One of these has empty

contents of the repositories. However, there are 8 more initial states in this model: the repository may have $0 \dots 5$ instances of `Entity1` and $0 \dots 4$ instances of `Entity2`.

Therefore, the requirements/constraints from the initial state are that:

- the entity multiplicities must be respected (Equation (3.17))
- the association multiplicities must be respected (Equation (3.18). (3.19))
- the links must be valid (Equation (3.20))

3.8 Properties of the transition system

Practically, the user of the DMDSL chooses the contents of the initial state. Thus, the goal is to prove that given a valid initial state, every transition should lead to a state where the invariants hold. If the invariants are violated, then one (or more) CRUD+A operations have not specified correctly. If the CRUD+A operations are correctly specified, then the specification is proven valid.

This proof can be done mathematically, or by using a tool. Here, the Alloy tool is used, as discussed in Chapter 4. Furthermore, Alloy also provides other benefits like automatic generation or completion of input models, and generation of test sequences. Therefore, the problem to solve in Alloy is, "Given a valid initial state for the DMDSL specification, is there some model M and corresponding sequence of transitions, that violates the repository consistency invariants?" The Alloy Analyzer creates models for the DMDSL specification, and looks for models and sequences of CRUD+A operations that invalidate the invariants. If no such model is found, then the specification is proven correct.

Chapter 4

Model exploration with Alloy

In this chapter, the semantics of DMDSL are defined in the Alloy specification language. The DMDSL is specified in Alloy with the goal of checking the correctness of the CRUD+A operations in terms of the repository consistency. Using specification tools to prove correctness of specifications has various merits and demerits. A practical merit is that no mathematical theorem proving is required, because using Alloy instead allows for explore the model and proving the properties automatically. Therefore, the formal specification in Chapter 3 is now translated into the Alloy syntax. For the reader unaware of Alloy, an introduction to the Alloy specification language is given in Section 4.2. An example specification is given in Section 4.2.6. Indeed, the textbook *Software Abstractions* [15] introduces Alloy with more comprehensive examples. Sections 4.3 and 4.4 denote the advantages and disadvantages of using Alloy for model exploration. The DMDSL specification in the Alloy syntax is presented in Section 4.5 - the syntax (Sec 4.5.1), transition system (Sec 4.5.3), CRUD+A operations (Sec 4.5.4), repository consistency (Sec 4.5.5), and model execution (Sec 4.5.6) are described. Additional remarks about the exploring the model in different ways are given in Sections 4.6 (desirable behavior) and 4.7 (model-based testing).

4.1 Requirements for a formal specification tool

There are specification tools such as mCRL2 [12], Event-B [1], TLA+ [27], Z3 theorem prover [6]. The requirements from the verification software are:

- Defining contents of states in the form of sets or lists, and the behavior of transition systems.
- Support for parametric transitions in transition systems.
- Ease of use with familiar set theory notation.
- Proven usage and familiarity in verifying DSLs in industrial environments.

4.2 Alloy Semantics: A Primer

Alloy as a specification tool can be used for verification of DMDSL specifications. Alloy has been used for verifying models and languages (practical Alloy usage, Section 6.1). It is possible to define contents of signatures and static semantics, and parametric predicates. The Alloy specification is expressed in relational logic and the syntax is generally easy This

section summarizes the syntax and semantics of Alloy as described in the book *Software Abstractions* [15].

4.2.1 Introduction

Alloy is a declarative formal specification language inspired by the Z-specification language and relational calculus. Alloy is used to describe structures and execute a **bounded exploration** of the structure through the Alloy Analyzer. An Alloy model[1] is a collection of constraints and equations, described on a set of structures.

In an Alloy model, *signatures* define the universe of the model, and each signature is a set. Each set can be populated with *atoms* – the atoms are generated by Alloy, but their multiplicity and relations are modeled by the user. A *relation* in Alloy maps a signature to one (or more) signature(s). It can be defined as a property of the domain signature, and the range of the relation may be the domain itself (e.g. the edge is a relation of the graph, and maps each vertex in the graph to 0 or more vertices). The Alloy semantics support binary relations out-of-the-box, and ternary relations can be implemented by using a library. The relations also have multiplicities, as explained later. A signature can be made *abstract* to create *child* signatures that inherit from the abstract signature.

Every statement in an Alloy model is a Boolean equation. A *fact* can be used to define constraints on the signatures in the model (i.e. equations that will always be true). A *predicate* can be used to define Boolean equations that can be referred to with an *identifier*, i.e. predicates are named (whereas facts are not named). An *assertion* can also be used to define Boolean equations, and may include references to predicates or other assertions. However, the converse is not true, and assertions cannot be used in a *run* command. Assertions can be referred to only with a *check* command (ref. Sec 4.2.4)

4.2.2 Signature and Relation Multiplicity

For addressing multiplicities, and quantifiers for predicate logic, Alloy includes quantifiers that correspond to predicate logic. The quantifiers for predicate logic are *no* ($\nexists$), *one* ($\exists$, unique), *lone* ($\nexists \cup \exists$), *some* ($\exists$, subset), *all* ($\forall$). As expected from the keywords - *no* expects 0 atoms, to be returned from the logical equation. Similarly, *one* expects 1 atom, *lone* expects 0 or 1 atoms, *some* expects >0 atoms and *all* expects all possible atoms. These expected multiplicities are translated into enforced multiplicities if used in a `fact`. Similarly, relations and signatures in Alloy also have a multiplicity. For signatures, the multiplicities are *lone*, *one*, *set*, *some*. The default is *set*, which ensures $>= 0$ atoms of the signature are created for analysis. The upper bound for this multiplicity is provided by the user. For relations, there are two multiplicities: one for the relation domain and one for the relation range. The former is derived from the signature that contains the relation, and the latter is given explicitly (*set* by default). For the range, the modeling can include any of the aforementioned signature multiplicities may be used, but also *seq* - where the relation range is an *ordered sequence*. Additionally, the *disj* keyword can be added, to make it disjoint among all atoms[2].

4.2.3 The small scope hypothesis

If an assertion is invalid, it probably has a small counterexample.

The Alloy reference (Section 5.1.3, [15]) introduces the *small-scope hypothesis*. Jackson notes

[1] AlloyTools: About Alloy

[2] The Alloy reference explains signature and relation multiplicity in detail.

in his text that *most bugs have small counterexamples* and that Alloy is useful for finding *subtle bugs* which do not need a large state space to be found. Jackson also states that the Alloy Analyzer works with a complementary role from a theorem prover – once the Analyzer fails to find a counterexample to an assertion, the specification **may be valid** for the bounded state space. Then, a theorem prover can be used to then prove the assertion correct for all scopes. Jackson suggests in the text that model exploration like Alloy and theorem proving should be used in a complementary way.

4.2.4 Executing Alloy models

Alloy treats a model as a conjunction of Boolean equations, which is possible only because all Alloy statements are Boolean equations. During analysis, these equations are formed into an SAT formula. SAT is the problem of determining if, for a given Boolean formula, there is a set of assignments to all variables in it such that the result is `TRUE`. Contrarily, to find a context where the formula is supposed to be always true, the formula can be negated, and SAT can be applied to find variable assignments where the formula evaluates to `FALSE`. It is well known that the SAT problem in predicate logic is undecidable[3].

An instance of an Alloy model is defined as the collection of atoms of signatures, within **given bounds**, that either satisfies or invalidates the SAT formula. Alloy is a bounded model exploration tool and based on its *small-scope hypothesis* (Sec 4.2.3), Alloy only explores models within the given bounds. To find any instance that satisfies the SAT formula, the *run* command is used. To verify that no instance invalidates the model, the *check* command is used. The *check* command, in essence, negates the SAT formula and tries to find any instance that satisfies it. Thus, it verifies that there are no instances where the input equations are violated.

The user might want to test the model for valid instances, or for (any) invalid instances. To execute the model, either the *run* or *check* command is used, which requires the equations to validate and the upper bounds for the model.
The *run* command finds *any valid instance* for which its input equations evaluate to `TRUE`. If the SAT cannot be satisfied, the input equations to *run* are determined inconsistent, i.e., there is no possible instance of the model that satisfies the predicates in the run command.
The *check* command finds *any invalid instance* of the model, where the input equation evaluates to `FALSE`. If no instance is found here, then the assertion **may be valid** for the given bounds. It might be possible that the input equations have contradictions. Such cases however, are easily found and can be discarded, because here *check* terminates unreasonably quickly.
The successful termination of the *check* command does not guarantee that the specification is valid: It is only guaranteed within the given bounds. However, it still builds confidence about the specifications, due to the small context assumption. But this does not guarantee the validity of the assertions an unbounded space.

4.2.5 Practical uses

Alloy is effective for exploring finite state spaces to find models, and is also suited for state-based formal specifications. Tools similar to Alloy are TLA+ [27], mCRL2 [12], Event-B [1], etc. The OCL[4] is similar to Alloy in terms of expressiveness. Although OCL has first-order

[3] Wiki: Horn clause and theorem proving
[4] Object Constraint Language, OCL

predicate logic, Alloy has a more powerful relational calculus and a simpler syntax[5].

Modeling transition systems is not straightforward in Alloy. Transition systems cannot be expressed as first class entities in Alloy but have to be modeled manually by using the `Ordering` library. This is also referenced in well-known examples of Alloy, the River crossing game and the Web attack. In both these examples, a state machine along with predicates specifying the transitions/dynamic behavior is described. To explore the state ordering, *first*, *next* and *last* keywords are available from the *ordering* library. The Alloy Java API is documented, which can be used to process of the generated outputs in the Analyzer.

4.2.6 Example

A brief example to show the syntax (and its meaning informally) of the Alloy language follows. In this example, a directed graph is defined. This graph has nodes and directed edges. There are braces following the signatures (Lines 4, 11) - these are called implicit facts and are applied on a signature itself. Implicit facts apply to each atom of the signature (keyword `this` refers to the atom) and are useful for defining binary relations (Line 4). From Line 4, the set `target_nodes`, for a given Node atom x, looks at the `Edge` signature where the source is `this`$=x$, and from this set extracts the `target` element, which must be of the expected type `set Node` as on Line 2.

The `Edge` signature (Lines 7-12) defines an edge, and also prohibits self-loops, i.e. a node may not have an edge to itself. The `Graph` signature is not used in model verification, but the *one* keyword here defines that only one graph is considered in model exploration. (Otherwise, the graph signature explores all permutations of Nodes and Edges. For a bound of 3, there are 3! = 6 graphs, of which 3 are explored!)

Consider that the goal is to ask Alloy to validate that the graph is acyclic. By definition, a node must not reach, *by any number of edges*, itself. To do this, the `NotCyclic` predicate, with the transitive closure operator `^`, is defined on Line 22. It denotes that for all nodes, the node n should not be found in the transitive closure of its `target_nodes` set. In this case, the goal is to look for any invalid models, so the `check` command is used on Line 25 with a signature upper-bound of 3. Alloy will explore models for signature sizes between 0 and 3, but by using the *exactly* keyword before 3, only the signature size of 3 is considered.

```
1  sig Node{
2      target_nodes: set Node    // relation
3  }{
4      target_nodes = { e: Edge | e.source = this }.target
5  }
6
7  sig Edge{
8      source: Node,
9      target: Node
10 }{
11     source != target
12 }
13
14 one sig Graph{
15     nodes: set Node,
16     edges: set Edge
17 }{
```

[5] https://alloytools.org/faq/how_is_alloy_related_to_ocl.html

```
18      edges = Edge
19  }
20
21  pred NotCyclic{
22      all n: Node | n not in n.^target_nodes
23  }
24
25  check { NotCyclic } for 3
```

Listing 4.1: A graph specification in Alloy, checked for acyclicity

Running this example in Alloy will generate a counterexample as given in Figure 4.1.

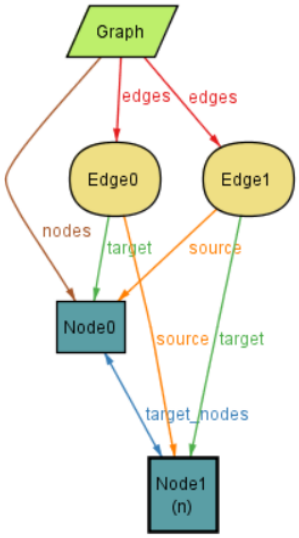

Figure 4.1: Counterexample for the graph cyclicity example

To troubleshoot the counterexample, the Evaluator feature of the Alloy Analyzer is used to view information. It becomes clear that the violation is because Node 0 and Node 1 point to each other, as verified by using the transitive closure in Figure 4.2. Note that the 3rd command is using the ^ operator. Thus, there is a cycle of length 1 in this counterexample.

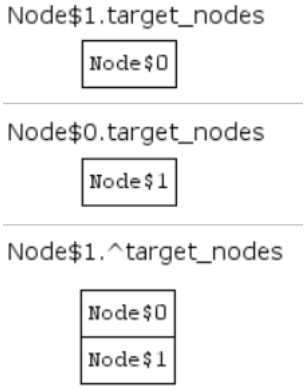

Figure 4.2: Example: Using Alloy Analyzer for checking cyclicity

4.3 Advantages of specification with Alloy

Exploring models with Alloy has various practical benefits:

- Practical in software engineering: Alloy is useful to define specifications and for given bounds, find valid models by using the *run* command. If it succeeds, the specification exhibits desired properties.
- Alloy is written in Java and available as a single `JAR` distributable, so it is platform-independent. The syntax of Alloy is simple to learn, and formal logic statements with set theory and predicate logic are easily translated into Alloy statements.
- *Expressiveness and Skolemization*: The Alloy language defines operators for transitive closure, `^ *`, on binary relations. This is part of the higher-order logic [17] semantics, that includes transitive closure, quantification over powersets, axioms on real numbers, etc., all of which are relevant to this text.
  Quantification over powersets must be enabled explicitly by configuring the 2nd-order Skolemization setting. In the DMDSL specification, the transitive closure is used for calculating cyclicity of associations, and for finding the connected instances in a set of links.
- Although the default settings do not enable this, Alloy can be configured to enable bounded recursion. This was also experimented with for the DMDSL formal semantics – but the transitive closure operator provides the required semantics and always terminates, while the recursion semantics do not.

4.4 Disadvantages of specification with Alloy

There are various limitations in verifying model specifications with Alloy:

- Exact length of traces: To get accurate results with Alloy, it is important to precisely set the bounds for model checking. In the `Ordering` library, the transition system has to be specified for an exact length. It can be possible that a long sequence of states ends up in a deadlock and the Analyzer does not report this.
- Scalability: Alloy also suffers from state space explosion, and depending on the complexity of the model, even increasing the state space bound by 1 can lead to significantly longer verification times. Similarly, reducing it by 1 can lead to missed counterexamples. Therefore, experimentation with the bounds is usually required and laboursome.
- In Alloy, type checking is evaluated globally, therefore the names of attributes, relations and predicates have to be defined carefully, to avoid scope and type resolution errors. Implementation hurdles like this can make it difficult to specify sets and relations, requiring effort to learn the Alloy notation.
- A practical limitation of implementation with Alloy is the deduction of failing conditions - if an assertion is violated and a counterexample is produced, *which equation caused the failure?* The Alloy Analyzer shows some related information but but for complex models, assertions in the Analyzer have to be manually checked.

4.5 DMDSL Semantics with Alloy

Given the formal specification of the DMDSL (Chapter 3), validating the correctness of the CRUD+A operations w.r.t. the repository consistency is the goal. Using a tool like Alloy to automatically explore the model and validate the consistency, is more practical than using theorem proving. Thus, here the the formal specification of DMDSL as expressed in Alloy. With the knowledge that Alloy supports relational logic which is stronger than first-order predicate logic and allows using transitive closure operator, the formal specification can be translated with the following strategy:

1. Translate all sets into signatures: the `Entity`, `Association`, `Instance`, and `Link`. The `Multiplicity` and `AssociationEndProperty` are also signatures here, that will be properties of the `Entity` and `Association`. For all these signatures, the well-formedness must be ensured.

2. Translate the static constraints (Sec 2.4) as facts. These facts must be respected by the Alloy Analyzer and will always hold true.

3. Then, define the `State` as a signature, so its contents (properties) can be modeled. As the goal is to model transitions, the `State` is defined as a sequence. Therefore, the `ordering` library is required to implement it – without using `ordering`, the atoms of `State` are disjoint/exclusive.
   The `ordering` library allows defining sequences between atoms a signature. The `first`, `last`, `next` properties of an ordering are useful to define transitions between elements. Therefore, a transition system can be defined and traversed by using the `ordering` library.

4. Translate the CRUD+A operations into predicates. Each predicate is a collection of Boolean equations, so the successful operations can be defined as predicates. In each CRUD+A operation, the relation between the given state and its next state is explicitly modeled.

5. To incorporate failure paths, another predicate may be used. This predicate is a logical disjunction of the successful predicate and each failure condition with the corresponding behavior of the transition system.

6. A fact *traces* can be specified, which defines the transitions between two states s and $s.next$ - using the *next* relation is possible because of the `ordering` library applied on the `State`. The traces defines the transitions between states as the CRUD+A operations.

7. Define the conditions of the initial state as a fact. Given this fact forces a valid initial state, the CRUD+A operations can be verified in terms of respecting the repository consistency invariants.

8. Translate the repository consistency invariants into predicates over `State` - the parameter is a state, and all the entities/associations must be checked for their multiplicities to be valid. If this predicate evaluates to true, then the repository in the valid in the state.

9. Define the expected model execution. The command used is *check{}*, with the model-checking bounds, and the *exact* length of the `State`. The input equations to *check{}* is

the repository consistency invariants. Thus, the Alloy analyzer can verify the correctness of the CRUD+A transitions from a valid initial state. With sufficiently well-defined CRUD+A operations, this verification should ideally produce no counterexamples.

4.5.1 Static DMDSL Semantics and Well-formedness Constraints

Defining the DMDSL static semantics into Alloy is straightforward for the sets. The Multiplicity, Entity, and Association constructs, as specified in the previous chapter, are translated as signatures. Corresponding properties like the *targets* relation, have been added based on their purpose. The *targets* is a relation mapping an entity to entities it can reach by associations, so this relation will be used in Alloy to define that the models are acyclic. Notice the similarity to the Acyclic Graph example in Listing 4.1 (Entity=Node, Association=Edge). To enforce an acyclic model, the transitive closure operator ^ is used.

```
sig Multiplicity{
    minimum: Int,
    maximum: Int
} {
    minimum >= 0 and maximum > 0 and maximum >= minimum
}

sig Entity {
    constructability: Bool,
    mutability: Bool,
    deleteability: Bool,
    multiplicity: one Multiplicity,
    targets: set Entity
} {
    targets = { a: Association | a.source = this }.target
}

sig AssociationEndProperty{
    cascade : Bool,
    multiplicity: Multiplicity
}

sig Association{
    source: Entity,
    target: Entity,
    sourceProperty: AssociationEndProperty,
    targetProperty: AssociationEndProperty
}
```

Listing 4.2: DMDSL formal model in the Alloy syntax

To define static constraints, the following fact is used:

```
fact constraints{
    StaticConstraints
}

pred StaticConstraints{
    no e: Entity | e in e.^associationTargets   // no cycles

    all e: Entity {
        (e.multiplicity.minimum = e.multiplicity.maximum) implies e.
            deleteability = False
```

```
10      }
11      // association properties: each line is a static constraint
12      all a: Association
13      {
14          a.sourceProperty.multiplicity.minimum = 0
15
16          a.target.deleteability = True implies (a.sourceProperty.cascade = True
                or a.targetProperty.multiplicity.minimum = 0)
17
18          a.sourceProperty.cascade = True implies (a.source.deleteability = True
                and a.source.multiplicity.minimum = 0)
19
20          a.sourceProperty.cascade = False implies a.source.mutability = True
21      }
22  }
```

Listing 4.3: Static constraints on the DMDSL model

The runtime structures, which are Instance and Links, are translated into Alloy as given below. The constraints defined under the signatures (on Lines 6,7,8,16,17), are used to enforce the well-formedness of the signatures. In the Instance, the `connectsSCD` and `connectsTCD` relations are used during deletion to build a graph of connected instances (exclusively for SCD/TCD-enabled associations).

The multiset Link from the formal specification is naturally translated to the `Link` signature (Line 11), and this definition in the default Alloy semantics allows distinct links to have the same content. This way, the *bag semantics* is easily captured by `Link`.

```
1   sig Instance {
2       type: Entity,                // the type() function
3       connectsSCD: set Instance,
4       connectsTCD: set Instance
5   }{
6       type.constructability = True
7       connectsSCD = { L: Link | L.link_target = this and L.link_type.
            sourceProperty.cascade = True }.link_source
8       connectsTCD = { L: Link | L.link_source = this and L.link_type.
            targetProperty.cascade = True }.link_target
9   }
10
11  sig Link {
12      link_source: Instance,       // L.s
13      link_type: Association,      // L.a
14      link_target: Instance        // L.t
15  } {
16      link_source.type = link_type.source
17      link_target.type = link_type.target
18  }
```

Listing 4.4: The signatures related to runtime

4.5.2 Definition of State in Alloy

The State in the Alloy model is defined as a signature. The transition system in Alloy is encoded as a sequence of `States`, by using the `ordering` library. The contents of a state can now be defined: the instances, repositories and links as a property of the state are defined

first (Lines 4,5,6). The `operation` is a simple indicator of the operation executed in the transition from the previous state, and the `output` relation presents the resulting success or error indication. The fact for the `State` signature (Line 10) specifies that the repositories is a subset of instances.

```
1  open util/ordering[State][language=Alloy, label=lst:AlloyInitState, caption={
       Definition of State in Alloy}]
2
3  sig State{
4      instances: set Instance,          // S.I
5      repositories: set Instance,       // S.REPO
6      links: set Link,                  // S.L
7      output: Output,       // output of CRUD operation of previous state
8      operation: Operation,             // CRUD operation of previous state
9  } {
10     repositories in instances
11 }
```

4.5.3 Initial State and Traces

To enforce the validity of an initial state, a fact is used. Here, the conditions in the initial state will specify a valid repository, so that the entities and association multiplicities are respected. The links in the initial state are forced to be valid as well.

The *traces* fact is significant: it allows defining the transitions possible in the transition system. From any state s in the system, *exactly one* of the create, add, update, or delete predicates can hold `TRUE`, so that a transition is made to the next state $s' = s.next$. Removing any one of the lines in the traces removes the CRUD+A operation from the possible transitions. Note that *each state in the Alloy model can perform a CRUD transition.* This is often a requirement in model-based testing with input-output transition systems (IOTS) [24].

```
1  fact traces
2  {
3      all s: State - last, s': s.next
4      {
5          some e: Entity, targets: set Link  | create [s, s', e, targets]
6          or some x: Instance  |  add[s, s', x]
7          or some y: Instance, targets: {set Link - y} | update[s, s', y,
               targets]
8          or some z: Instance  | delete[s, s', z]
9      }
10 }
11
12 fact first_state{   // 'first' is provided by the ordering library, as the
       entry point of the ordering
13     first.operation = Initialize
14     first.output = Success
15
16     Entity.multiplicity.minimum = {0} implies (no first.instances)
17
18     all e: Entity | EntityMultiplicity[first, e]
19
20     all a: Association  | AssociationMultiplicity[first, a]
21
22     all p: first.links {
23         p.link_source in first.instances and p.link_target in first.instances
24
```

```
25          p.link_source in first.repositories implies p.link_target in first.
                repositories
26      }
27  }
```

Listing 4.5: Defining initial state and traces with Alloy

4.5.4 CRUD operations

Based on the traces fact, it is specified that a CRUD+A operation is possible in a state s in the Alloy model. The strategy to encode CRUD+A operations is: first, define a predicate that encodes the *happy flow* - when the requested operation can be *performed and succeeded*. Then, a predicate is defined for every possible violation of preconditions of the successful transition, where the operation is *performed but failed*.

To encode a successful CRUD operation, the parameters are two states s and s', which are ensured to be the current state and next state by the fact *traces*. As a simple example, the Add operation is given as a predicate. It expects that an instance is already created and its associations should be satisfied, after which the instance can be added to repository. Note that the definition of fact *traces* specifies `add` and not `successful_add`.

```
1   pred successful_add[s: State, s': State, i_to_add: Instance ] {
2       // preconditions
3       i_to_add in s.instances
4       i_to_add not in s.repositories
5       all p: s.links | p.link_source = i_to_add implies p.link_target in s.
            repositories
6
7       // relation of state s and s' -- The traces ensures that s' is the next
            state of s
8       s'.instances = s.instances
9       s'.repositories = s.repositories + i_to_add
10      s'.links = s.links
11  }
12
13  pred add[s: State, s': State, x: Instance ] {
14      s'.operation = Add
15      {
16          successful_add[s, s', x]
17          s'.output = Success
18      }
19      or
20      {
21          x in s.repositories
22          states_are_equivalent[s, s']        // contents of states do not
                change!
23          s'.output = AlreadyInRepository
24      }
25      or ...
26  }
27
28  pred states_are_equivalent[s: State, s': State]{
29      s.instances = s'.instances
30      s.repositories = s'.repositories
31      s.links = s'.links
32  }
```

Listing 4.6: The add transition as a predicate in Alloy

The successful (happy-flow) predicates to define the Create, Update, Delete operations are given in Appendix A.

4.5.5 Repository Consistency

Following, are predicates to check the entity and association multiplicity for the respective parameters (some given `State` and `Entity`/`Association`). The `#(...)` operator is used for set cardinality, which is predefined in the Alloy syntax.

```
pred EntityMultiplicity[s: State, e: Entity] {
    e.multiplicity.minimum <= #(instancesOfType[s.repositories, e]
    #(instancesOfType[s.instances, e] <= e.multiplicity.maximum
}

pred AssociationMultiplicity[s: State, a: Association] {
    am_source[s, a]
    am_target[s, a]
}

pred am_source[s: State, A: Association]{
    all x: instancesOfType[s.repositories, A.target]
    {
        #(incomingLinksOfType[s.links, A, x]) <= A.sourceProperty.multiplicity
            .maximum
    }
}

pred am_target[s: State, A: Association]{
    all x: instancesOfType[s.repositories, A.source]
    {
        let target_links = outgoingLinksOfType[s.links, x, A]
        {
            target_links.link_target in s.repositories
            #(target_links) >= A.targetProperty.multiplicity.minimum
            #(target_links) <= A.targetProperty.multiplicity.maximum
        }
    }
}
```

Listing 4.7: Translating consistency predicates to Alloy

4.5.6 Executing the model

To run the Alloy model, the check command is used along with the repository consistency invariants as the input. In this example, the signatures are limited to a bound of 3 and there are 8 states in the transition system. The code here represents the intention of checking the semantics of the CRUD+A operations. Given the transition system for some model m (that the Alloy Analyzer determines), each transition is one of the CRUD+A operations. The `check {Consistency}` command verifies that these transitions do not violate the invariants in any state of the transition system. Thus, this code is used to answer the primary goal of this work: Are the semantics specified correctly such that the invariants cannot be violated for any model m? If the `check {Consistency}` command does not produce any counterexamples, then the intended semantics are specified correctly. In that case, no static constraints are required, and no refinements to the CRUD+A operations are required. For every counterexamples produced, the semantics are refined.

```
pred Invariants{
    all s: State {
        all e: Entity | EntityMultiplicity[s, e]
        all a: Association | AssociationMultiplicity[s, a]
        all p: s.links {
            p.link_source in s.instances and p.link_target in s.instances

            p.link_source in s.repositories implies p.link_target in s.
                repositories
        }
}

pred Consistency{
    Invariants
}

check { Consistency } for 3 but exactly 8 State
```

Listing 4.8: Model-checking DMDSL specification in Alloy

4.6 Checking for desired behavior with Alloy

The Alloy Analyzer only checks the *explicitly provided equations* for finding counterexamples. Thus far, it has been the `Invariants` that define the repository consistency. However, it may be useful to define predicates that reflect desirable behavior. This maybe something like: If an Entity has the Constructable property, it should be successfully constructed at least once, and vice versa for its deletion. Therefore, the Alloy Analyzer can query for this desirable behavior in the DMDSL specification, and for this the following changes are added to the specification:

```
pred Liveness{
    // all entities that are constructable or deleteable, will have the create
        /delete operation happen at least once

    all e: Entity {
        e.constructability = True implies some s: State, links: set Link |
            create[s, s.next, e, links] and s.next.output = Success

        e.deleteability = True implies some s: State, x: Instance |  x.type =
            e and delete[s, s.next, x ] and s.next.output = Success
    }
}

pred Consistency{
    Invariants
    Liveness            // another property is added to the input
}

check { Consistency } for 3 but exactly 9 State // verification
```

Listing 4.9: Checking for desirable behavior with Alloy

4.7 Using Alloy for Model Testing

It is possible to have a more model-based approach to exploring the DMDSL semantics in Alloy. In this context, input test models can be specified, and the *run* command can be used instead of *check*. As a result, this enables testing of the DMDSL specifications (as a transition

system), for a given model(s) that satisfy some properties. The properties can be specified in Alloy with predicates, where one (or more) of the following can be applied on a model:

- Explore the model only if it has properties (`model_test_properties`) such as a number of entities and associations.
- Explore the model only if it includes a set of expected CRUD+A operations (`model_test_traces`): for example, at least two successful deletion operations.
- Explore the model only if it corresponds to an expected final ("last") state with some properties: for example, the final state should have at least 2 repositories and links.

The *run* command can be used to see if the conjunction of the Invariants and the `model_under_test` leads to a valid instance of the model. Thus, the Alloy Analyzer can find test sequences that satisfy all the conditions, and these sequences can be translated to function calls over the generated code. The sequence also represents the expected behavior of the implementation, and it can be translated into C++ code and executed on the implementation.

An example is given below:

```
1  pred model_under_test{
2      model_test_properties
3      model_test_traces
4      model_test_last_state
5  }
6
7  pred model_test_traces{
8      some s: State | s.operation = Create and s.next.output = Success
9  }
10
11 pred model_test_properties{
12     #Entity >= 1   and  #Entity <= 4
13     #Association >= 1 and #Association <= 2
14     #Instance >= 1 and #Instance <= 5
15     #Link >= 1 and #Link <= 5
16
17     some p, q: Association | p.target = q.source
18 }
19
20 pred model_test_last_state{
21     #(last.instances)  >=  5   // last state should have >4 instances
22     #(last.prev.links - last.links )  >=  1    // delete 1 at least link in
           last state
23 }
24
25 pred Consistency{
26     Invariants
27 }
28
29 run { Consistency and model_under_test } for 4 but exactly 8 State
```

Listing 4.10: Model-supported testing to seek valid instances with Alloy

4.8 Conclusions

This chapter introduces the DMDSL specification given in Alloy. There are various advantages of using Alloy: it can automatically explore models possible in the specification and generate

test sequences (traces), counterexamples (in theory) are of a small context. The disadvantage is that the transition system has to be modeled manually, furthermore the length of the transitions must be specific and the generated counterexample must be manually examined for every assertion.

Defining the invariants of repository consistency allows using the `check{}` command and verifying that there is no state where the entity or association multiplicities are violated. By defining a fact that constraints the transitions between states to be one of the CRUD+A operations, this verification validates that the CRUD+A operations do not violate the repository consistency. Alternatively, it is also possible to define model instances that should be tested with Alloy, and generate test sequences on the models that can in the future used for model-based testing.

Chapter 5

Results

The semantics of the DMDSL have been specified in Alloy. These specifications can now be checked for correctness (as in Section 4.5.6). To test the specification, two approaches were taken. First, an initial state was not forced, with the goal to find models where there is no initial state: These are models that should be invalidated from code generation, by adding static constraints. The results are presented in Section 5.1. Then, the CRUD+A operations are verified, for which the initial state conditions (as in Sec 4.5.3) are enforced. Given the valid initial state, the CRUD+A operations, and the definition of repository consistency, the dynamic semantics can be validated. The results are given in Section 5.2. Experimentally, it was derived that including the Update operation as a transition significantly increases verification time. This is explained in Section 5.2.2. Finally, another disadvantage of Alloy is explained in Section 5.2.3, documenting deadlocks found in bounded model checking with Alloy - these are not deadlocks in the transition system semantics, but only in the bounded exploration of Alloy. This highlights the awareness required by the Alloy user to get a relevant model exploration results.

5.1 Models without an initial state

In this analysis, the DMDSL specification is not forced to have an initial state. This provides insights into additional constraints to evaluate validity of models. Once the initial state can be ensured valid, the dynamic semantics can be verified. This section presents models without an initial state. Note that these are reported for v3.6.0 of the ASOME DMDSL.

5.1.1 Cyclicity of Associations

The ASOME v3.6.0 implementation prohibits the user from creating an association to the same Entity, but does not prevent cycles like the one shown in Figure 5.1. This model has a valid initial state with 0 instances, but there is no next state since no CRUD+A operation can be done on this model. This is because the creation of the instance of Entity0 requires exactly 1 instance of Entity1 to exist following Association0, but Entity1 requires an instance of Entity0, following Association1.

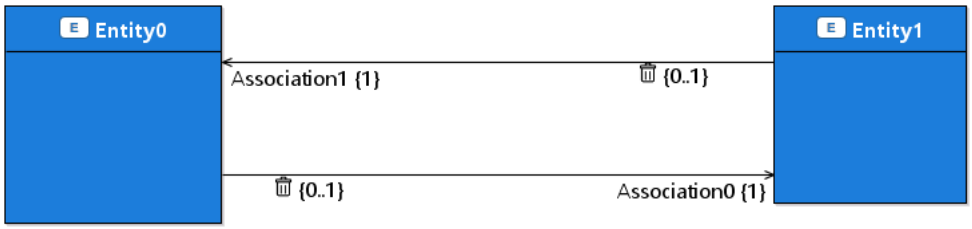

Figure 5.1: Cyclic relations in the implementation

Solution: The cyclicity detection in the DMDSL implementation should be extended to detect cycles of length n.

5.1.2 Association Targets

Consider Figure 5.2. This counterexample has no valid initial state because the creation of the instance of Entity1 requires exactly 1 instance of Entity2 to exist following the association 'entity2'. However, Entity1 is Unconstructable, so for the considered DMDSL specifications that exclude Realization semantics, it will never be possible to instantiate Entity1. Therefore, it will never be possible to instantiate Entity0.

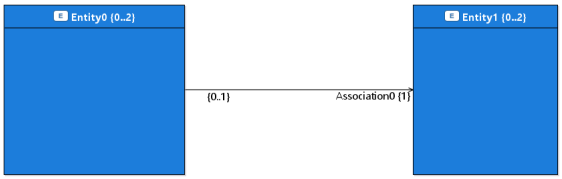

Figure 5.2: Association that is never realized

Solution: A new constraint may be provided: A target of an Association should always be Constructable, if it has a minimum multiplicity of 0.

5.1.3 Non-causal associations

The model given below should be invalid because the initial state expects at least 1 instance of `Entity1` via constructor delegates, but `Entity1` has a target `Entity2` which has 0 instances initially. Therefore, the initial state is invalid since it cannot satisfy the invariants for entity multiplicity.

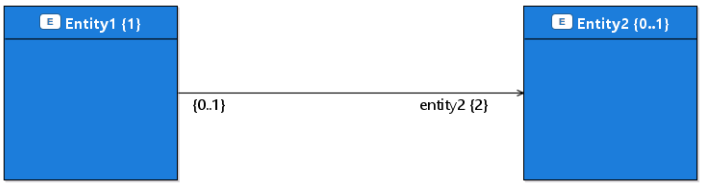

Figure 5.3: Invalid due to multiplicity

Solution: In the constructor delegates, it is made possible to create instances of any entity

that is Constructable. Therefore, the initial state can have >0 instances for any entity that is Constructable.

5.1.4 Miscellaneous Counterexamples

Some counterexamples were discovered during the understanding of the DMDSL semantics of inheritance, without the formal specification.

Specialization-Association (cycle of length 2)

In the figure below, there is an association from `Entity1` to `Entity2`. There is also an association from `Entity2` to `Entity3`. This model is so far valid, but consider the relation that `Entity3` is a child of `Entity1`, and inherits the association to `Entity2`. This is again a cyclic model that the validation should reject.

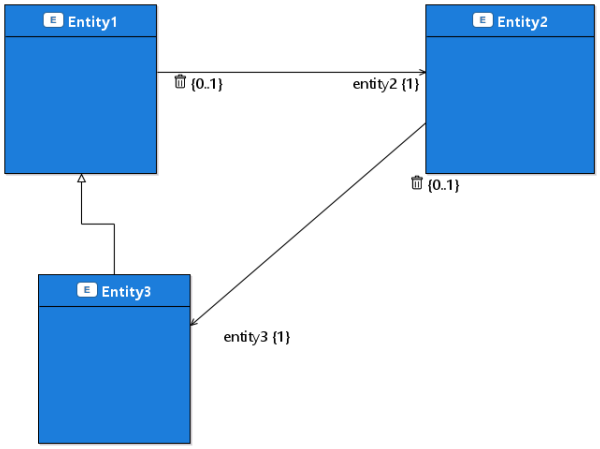

Figure 5.4: Specialization relation creates cycles

Solution: The cyclicity detection has to be modified, to check for loops after the evaluation all the inheritance relations.

Association to specialized (cycle of length 1)

In the figure below, there is an association from `Entity1` to `Entity2`. There is also a specialization from `Entity1` to `Entity2`, so `Entity2` is specialized and inherits all associations of `Entity1`. This introduces cyclicity so the model is invalid.

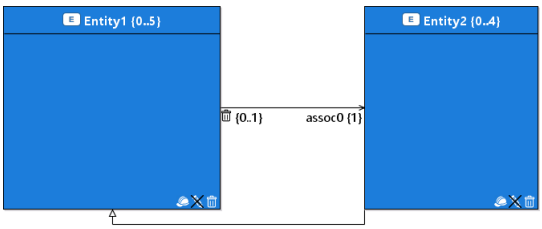

Figure 5.5: A cyclic association-specialization

Solution: The cyclicity detection has to be modified if specialization is taken into account.

5.2 Verifying dynamic semantics (Valid Initial State)

For verification of the repository consistency and dynamic semantics of DMDSL, the DMDSL specification is forced to have a valid initial state respecting the repository consistency. Given the valid initial state, the goal is to check with Alloy if there are any models (and corresponding CRUD traces) which lead to an invalid state. Finally, **bounded verification using the Alloy analyzer in this text does not find a violation of repository consistency invariants**. The analyzer reports the number of primary vars and clauses in the SAT formula that it will validate, which in this case, is the repository consistency equations.

Experimental Setup

The experiments in Alloy are executed on a laptop device (Lenovo Yoga) with Intel i7-8500U (2.0GHz base, 3.8GHz turbo), having 8GB of memory. The Alloy Analyzer executable runs tests on 1-core, 2 threads. The settings in Alloy are: maximum memory = 3072MB, maximum stack: 32768K, solver: SAT4J, Skolem depth: 2.

5.2.1 CRUD Traces: Correctness

It was concluded by the Alloy *check* command that the intended specifications of the Create, Add, Update, and Delete actions are correct with respect to repository consistency. That is, given a valid initial state, carrying out any sequence of CRUD+A actions as defined in Alloy, does not violate the repository consistency. This can be claimed with a level of confidence that the testing is in a bounded state space.

In the table below, the bounds and the verification time for the Alloy experiments are presented. Evidently, increases in the bounds and state space explosion significantly increase the verification time. This shows the lack of scalability in checking large models with Alloy.

Signatures bound	State/Traces bound	Primary Vars	Clauses	Verification time
3	6	708	129376	1132.98 seconds (19 minutes)
3	7	821	146211	6347.21 seconds (105 minutes)
3	8	946	163132	321 minutes
4	6	956	232322	92 minutes
4	7	1098	262616	9 hours
4	8	1256	293770	>12 hours

Table 5.1: Verification time for DMDSL in Alloy - CRUD+A operations

The lack of violations shows that the formulated well-formedness constraints ensure consistent repository behavior at runtime. Given the correctness of the semantics, the specifications and the CRUD+A operations defined in Alloy give the language engineers insights to the required preconditions and semantics in the DMDSL implementation, so that the implementation guarantees repository consistency.

5.2.2 Traces without the Update operation

It was found by experiments that including the Update operation as a possible transition in the traces significantly increased the time in verification. The model cannot be verified if the Create or Add operation is removed as they are crucial. However, Update can be removed from verification as it modifies an existing instance and this can be repeated ad infinitum. The Delete operation also has an interesting case where the Alloy Analyzer finds sequences of

create, add, and delete of the same instance as long as possible, thus both these operations increase the model verification time significantly. To check the Update operation for correctness, the Delete operation can be removed as a possible transition. After the Update operation is verified to produce no counterexamples, it is removed as a possible transition so that the Delete operation can be included, as it is much more significant to the specifications. This step is also grounded in reality, since the recommended modeling practice for DMDSL users is to use immutable entities wherever possible (thus the Update operation is not encountered often).

Intuition and experiments show that it is likely due to the update operation that the CRUD+A operations take significantly longer to verify for increases the time because of the repeatable nature of an update. For example, refer the figure below. In `State5`, the link `Link1`, pointing from source `Instance1`, has been existing. In `State6`, the `Link1` is replaced with `Link0` by an update of `instance1`, i.e. the association gets updated. In `State7`, the link can be updated to `Link1` again and this can be repeated for the rest of the transitions.
Regardless of the contents of the links, such repeatable behavior may be causing the analyzer to take significantly longer, compared to Create-Add-Delete. Naturally, it also causes Create-Add1Update-Delete to take significantly longer to verify, in fact the checker has to be terminated after 12 hours for a small bound of 4 with 8 States (Table 5.1).

this/Link	link_source	link_type	link_target
Link$0	Instance$1	Association$1	Instance$3
Link$1	Instance$1	Association$1	Instance$3

State$5	Instance$0	Instance$1	Link$1	Update$0	Success$0
	Instance$1	Instance$2			
	Instance$2	Instance$3			
	Instance$3				
State$6	Instance$0	Instance$1	Link$0	Update$0	Success$0
	Instance$1	Instance$2			
	Instance$2	Instance$3			
	Instance$3				

Figure 5.6: Update repeating sequences

Therefore, the specification is tested without the Update operation. In this case, only the Create-Add-Delete operations are possible and the following verification times are recorded corresponding to the bounds.

Signatures bound	State length	Primary Variables	Clauses	Verification time
3	6	660	79760	16 minutes
3	7	773	90920	36 minutes
3	8	946	163132	58 minutes
4	6	956	232322	124 minutes
4	7	1098	262616	298 minutes
4	8	1256	261523	448 minutes

Table 5.2: Verification time for DMDSL in Alloy - Create/Add/Delete operations

As a direct comparison, Table 5.3 below shows the time difference in adding the Update and Delete transitions.

Traces	Signatures Bound	State Space	Primary Vars	Clauses	Verification Time
Create-Add-Delete	4	8	1256	264233	17,924.82 sec (5 hours)
Create-Add-Update	4	8	1256	289276	26,912.370 sec (7.5 hours)
Create-Add-Update-Delete	4	8	1256	297770	>12 hours (terminated)

Table 5.3: Comparing the CAD-CAU-CAUD operations in traces

5.2.3 Deadlock scenarios in Alloy with bounded verification

In some situations, it is possible to encounter deadlocks, especially so in bounded model checking. If the traces include only the create-add operations then it is easy to encounter a deadlock: a situation where a valid state is reached, but no successful create-add operation is possible. Consider the bounded state space of 3. Therefore, the 3 instances can exist by Create, and stored to repositories by Add. This is 6 operations. With an empty initial state, this can allow for at most 7 states. If the Alloy analyzer is asked for a state space of 8, the following can be seen:

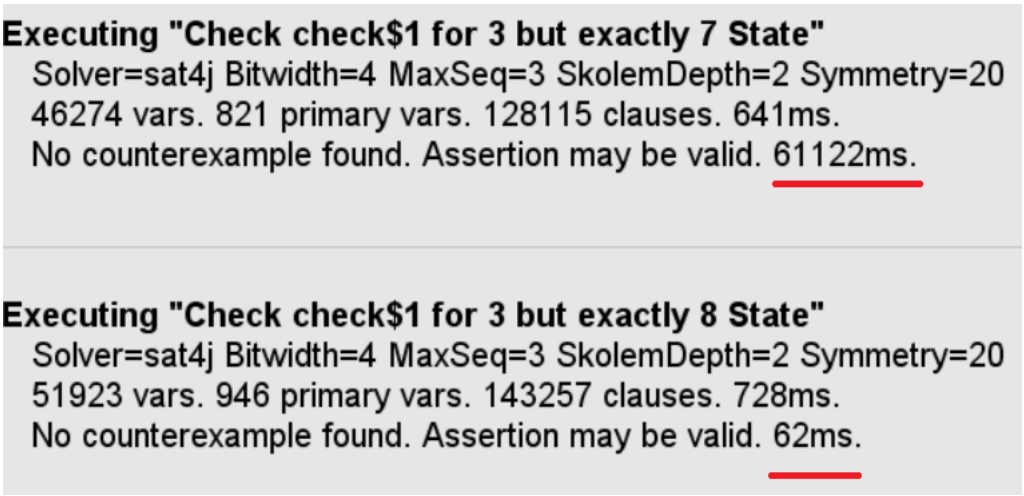

Figure 5.7: Possibility of deadlocks in bounded checking

The verification for a length of 8 returns immediately and is a logical error not reported by

Alloy, which the user needs to spot (or specify as a predicate in Alloy, i.e. deadlock detection). Here, the Analyzer locates a deadlock after the 7th state: No more creations or additions are possible. In this context, the initial state has empty contents, thus, once the 3 instances have been created and stored in States 1-7, there is no possible operation to make into in the last (8th) state, as deleting is not a possible transition, and the preconditions for successful creation are not satisfied.

5.3 Conclusions and insights

The experimentation with Alloy and examining the counterexamples is a thorough and thought-provoking process. Such an exercise forces the language developers to criticize and reflect on the language: the formal syntax of models, well-formedness constraints, and the dynamic semantics. The analysis coerces language developers to interpret counter-intuitive model examples and traces generated by a formal verification tool, resulting in refinement of the language semantics and fewer bugs.

In terms of testing DMDSL with Alloy, the issue of scalability is seen early. While there are factors (hardware capability, other processes taking up resources, low number of samples) that affect the confidence on the verification time with Alloy, the exponential increase in time taken with the size of the transition system is evidence to the limited scalability of Alloy for verifying dynamic semantics. Furthermore, increasing the bound of the signatures (from 3 to 4) already introduces a verification time of >12 hours. The results from the experiments on DMDSL specifications show that the CRUD+A operations have been specified correctly in Alloy - with confidence in a bounded state space. The DMDSL implementation can manually be compared to the Alloy specification by the language engineers, so that the implementation can conform to the formal specification and thus be guaranteed to ensure the repository consistency.

In the real-world, there is a trade-off between man-hours and machine-hours invested in testing a DSL. The DSL engineers involved in this text, agree that letting a formal verification tool check the language specifications for correctness may cost machine-hours but as a result, man-hours can be saved which is more valuable. Invalid models that appear later when the DSL is tested manually, can already be found early-on in formal verification. Indeed, the language development team would require a formal methods expert, but the automated verification of the language specifications produces more value compared invested man-hours in testing. The added value is reflection and criticism of the DSL semantics, and the early detection of invalid models leading to confidence in the well-formedness constraints.

Chapter 6

Discussion

6.1 Related Work

There is work related to model-based testing of code generators in industry. Frenken et al. [9] in a team of 3, focuses on verification of a DSL by using MBT and specifying the DSL as a labeled transition system. In their work, the formal specifications of the language OIL at Océ are translated in mCRL2 [12], and tested with JTorX [2]. The language models the control behavior and transitions between states, i.e. it is a transition-based specification. Their work uses an input-output transition system (IOTS) as proposed by Tretmans [24], that introduced the *input-output conformance (ioco)* theory for model-based testing. An adapter converts the MBT traces to C++ logic, and executes against the C++ implementation. The works of Frenken and Tretmans were on a transition-based spec for a given DSL model, (and not on finding invalid models).

Sijtema et al. [21] presents an experience of formal methods engineering in the industry, where they define a model-based specification and testing of a software bus implementation. In their work, the implementation is developed along with the formal specification in an Agile setting. They conclude that this way of developing leads to fewer bugs in the implementation. They specify the model in mCRL2 and apply model-based testing with JTorX. They report that MBT is more comprehensive and discovers bugs that are not seen in conventional testing. Additionally, model-checking reports that MBT can still be less optimized and even incomplete, if requirements pertaining to bad-weather behavior (exceptions, unexpected inputs, etc.) are missing. They report that adding model-checking is expensive in terms of time but leads to almost 100% model coverage.

In this text, MBT was not applied although it was one of the goals of the project. The focus was on formalizing the DMDSL semantics and checking some of its properties: namely, the repository consistency, well-formedness constraints and the CRUD+A operations. It has been shown how Alloy can be used for generating test input models and test traces. This is done not in an exhaustive way like the MBT tools in related literature, and is a future work from this project. Proving the consistency of the repositories given the formal specification can be checked with a theorem prover but this require a different tool such as Z3 [6].

The work of Cunha et al. [4] provides insights into translating UML diagrams annotated with

OCL into Alloy specifications for verification. Shah et al. [20] as well introduces a model transformation from Alloy instances to UML class diagrams with the UML2Alloy transformation. This gives insights into translating static constraints in Alloy to OCL. Alloy has also been used in practice, for supporting the development of domain-specific languages. Gammaitoni et al. [10] illustrates the usage of Lightning and the F-Alloy model transformation language (introduced by Lightning) to develop a domain specific language. They also present how using Alloy for automatic analysis of the language allows the language developers to incrementally validate it and get visual feedback of the language. The work of Erata et al. [8] introduces AlloyInEcore, a tool to specify metamodels with their static semantics to facilitate automated formal reasoning on models. Erata et al. [8] states that AlloyInEcore allows the user to specify meta-models with their static semantics and automatically detect inconsistent models as well as complete partial models. They apply Alloy on industrial use cases (DSL in automotive domain), but on the static semantics of models. Furthermore, they conclude that verification of static semantics With Alloy is viable, however the verification of dynamic semantics is left as a future work, which is the direction that this thesis approaches.

6.2 Future Directions

This thesis aimed at introducing a subset of the formal specifications of the DMDSL static and dynamic semantics, and developing a bounded-model checking solution for it with the Alloy specification language. With the complete formal specifications of DMDSL, it is possible to comprehensively assert the correctness of the implementation. However, for this purpose Alloy may not be suited as it is evidently does not scale well for dynamic semantics of industrial complexity use cases. More powerful software like mCRL2 might be beneficial. With this direction, the complete formal specification of DMDSL can be specified and verified, which would a beneficial trade-off for reduced man-hours against machine-hours.

From the Alloy Java API, it is possible to capture the information generated by the Analyzer. In this case the required information would be the traces between states and the entities/associations. This can be used to introduce a more model-based testing approach with Alloy, and this API can be useful to pass inputs to an adapter that can translate the CRUD traces into C++ code.

Chapter 7

Conclusions

The DMDSL developed by Altran and ASML is a mature, and complex industrial use case. The ASOME software has been in use since 2015 and the languages in it have evolved to support new features. Because the ASOME is described informally in text and diagrams, its precise semantics are unknown. Over the years, language developers have manually tested ASOME and added well-formedness constraints to invalidate models that have inconsistent behavior in the repository. From a software engineering perspective, the correctness of the ASOME semantics is not guaranteed and requires verification.

This text helps in understanding the relevance of incorporating formal methods during the development of languages. The DMDSL is a language in the ASOME software. It has static and dynamic semantics that introduce data objects like entities having properties and relations (associations) between them. The entity data objects are stored in repositories, to commit them to memory. The entities and associations also have multiplicities which should be respected. In the dynamic semantics, the entities and associations are instantiated by the CRUD+A operations. Finally, the goal is to verify that the multiplicities are respected in the dynamic semantics. This thesis addresses this by introducing the formal specification for a subset of the DMDSL semantics (entities and associations); and it verifies the correctness of the dynamic semantics (CRUD+A operations) in terms of the repository consistency. The dynamic semantics is defined as a transition system where the states capture the content of the repositories, and the transitions are the requested CRUD+A operations.

The static semantics are introduced as sets in the formal specification. Well-formedness constraints are used to define valid contents of the sets, and reject invalid models. The specification requires modeling the contents of a state - the set of created instances, set of instances stored in repository, and set of links between existing instances. The dynamic semantics are introduced that allow CRUD+A operations between neighboring states. The definition of an invariant expresses the criteria for correctness: the repository consistency. These invariants define that in every state, the entity and association multiplicities should be respected.

By using Alloy as a verification tool, this work explores the specifications of the DMDSL for the repository consistency. A number of corner cases are uncovered with no initial state, and no instances or links can be created in these transition systems. Static constraints are defined so that a model should always be well-formed and possible to initialize. To verify the dynamic

semantics (CRUD+A operations), constraints are introduced on the initial state. Through Alloy, it was verified that the intended specifications of the CRUD+A operations are correct w.r.t. the repository consistency.

A few limitations are also encountered in using the Alloy specification language. Of course, SAT solving is a well-known undecidable problem, consequently larger state spaces cause Alloy to suffer from state space explosion and limited scalability. Troubleshooting counterexamples for complex models is difficult as it is done by manually inspecting predicates (although, using the Alloy Java API can simplify troubleshooting). Significantly, the bounded model-finding also requires an exact number of states in the transition system - considering the limited scalability that the experiments show, the bounds for model finding have to be carefully selected.

On the other hand, with the small state hypothesis, using Alloy provides higher precision in testing for a limited investment in time and testing effort. Developing the intended formal DMDSL specification, along with its implementation, would also mean initially a simpler formal model that co-evolves with the implementation. In the context of Alloy, this would likely still lead to a limited scalability situation very early. However, using a formal methods way of developing leads to better quality software, as supported by literature [8–10,21]. In this context, using formal methods for verification of DMDSL during its development would cause the language developers to reflect on the semantics and handle counter-intuitive examples early on, rather than discovering issues while testing the generated code.

7.1 Answers to the research questions

RQ1. How should the semantics of DMDSL be precisely specified, considering the informally defined semantics?

Based on discussions with the language developers and informal references, the semantics of the legacy code generator were formally specified. With such a process, the formal specifications can be noted down and then specified in some tool. Simply noting down the formal specification based on discussions already introduced new insights and various questions about the intended semantics of DMDSL and the semantics in the implementation. In the general perspective of software engineering, development of the formal specification along with the implementation can allow for co-evolution of both, and tool-based verification can provide benefits of automated model exploration and reduced man-hours in testing, while improving the precision of the language semantics.

RQ2. What is a suitable tool for formal verification of DMDSL semantics, and what is required?

There are various tools suited for formal verification - Alloy, mCRL2, Event-B, Z3-prover are some examples. The requirements are that, based on set theory and predicate logic, the semantics of DMDSL can be specified and automatically explored. Additionally, it should be possible to define a transition system. The contents of a state are significant and should be defined, as well as the changes in the contents should only be possible by transitions between distinct states. Alloy allows specifying parametric transitions, so that CRUD+A operations on arbitrary objects is possible. Alloy satisfies *most* requirements, however the transition system must be specified manually (as a positive, the model can be explored automatically). Alloy has also been used in industrial context for developing and verifying DSLs, so it seems to be a

suitable candidate to examine for this work. Thus, based on the requirements, Alloy is selected as a tool for verification of DMDSL specifications. The translation of formal specifications and CRUD operations into Alloy syntax is fairly straightforward and allows for verifying the correctness of the CRUD+A operations in terms of repository consistency. Alloy also supports deriving test cases for the code generator in two ways: (1) by automatically generating input models for DMDSL (Section 4.5.6), and (2) by creating a sequence of transitions that can be translated to C++ calls against the generated code (Section 4.7). This however, is different from the way dedicated MBT tools function.

RQ3. What are the benefits and disadvantages of a formal verification approach in a real-world industrial context like DMDSL?

Using a specification tool like Alloy introduces formal verification in the development cycle, and enables the language developers to see cases of counter-intuitive models that the implementation might be considering valid. Often the counterexamples are small and can point to fixable flaws in the semantics, such as cyclicity or multiplicity errors as seen in this work, or by simply adding well-formedness constraints that prevent the model from validating.

Alloy is free and open-source, and has a readable but tricky syntax to adopt, translating from set theory. Alloy usually provides small counterexamples with the small scope hypothesis. The benefits of using Alloy are that by assuming a valid initial state the specified CRUD+A operations can be proven correct. Alternatively, if an initial state is not forced, Alloy can even produce counterexample models that cannot be initialized. Because of its small-scope hypothesis, the counterexamples are often small and can be easily understood by examining them. Therefore, using an automated model finding approach with Alloy allows trading off machine-hours in favor of reduced man-hours.

The disadvantages of Alloy are limited scalability, limited presentation and visualization of the assertions, and furthermore minor implementation or semantics-related issues in Alloy make it tricky to master. Finally, the work in this text does conclude that while Alloy struggles in verifying large state spaces of complex dynamic semantics (i.e. transition systems and multiple invariants), a more suited tool for modeling transition systems can be useful to extend the DMDSL specification and verification of dynamic semantics.

Experiences with learning Alloy: The Alloy specification language is easy to learn and start using, especially for exploration of static models. However, it has a learning curve that, with the hurdle of manually modeling transition systems, can be steep. (The curve might have seemed steeper than normal because the author was struggled with formal spec and declarative reasoning.) The tutorials in the Alloy documentation helped in adapting the syntax and transition systems, and modeling the DMDSL specification allowed the author to understand the intricacies of Alloy.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010. 34, 36

[2] Axel Belinfante. Jtorx: A tool for on-line model-driven test derivation and execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–270. Springer, 2010. 56

[3] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Software*, 24(5):48–55, Sep. 2007. 2

[4] Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software & Systems Modeling*, 14(1):5–25, 2015. 56

[5] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003. 2

[6] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. 34, 56

[7] Doron Drusinsky. *Modeling and verification using UML statecharts: a working guide to reactive system design, Runtime Monitoring and Execution-based Model Checking*. Elsevier, 2011. 3

[8] Ferhat Erata, Arda Goknil, Ivan Kurtev, and Bedir Tekinerdogan. Alloyinecore: Embedding of first-order relational logic into meta-object facility for automated model reasoning. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 920–923. Association for Computing Machinery, 2018. 57, 59

[9] Mark Frenken, Tim AC Willemse, Louis van Gool Océ, Olav Bunte, and Jasper Denkers. Code generation and model-based testing in context of oil. Master's thesis, Technische Universiteit Eindhoven, 2019. 56, 59

[10] Loïc Gammaitoni, Pierre Kelsen, and Christian Glodt. Designing languages using lightning. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, page 77–82, New York, NY, USA, 2015. Association for Computing Machinery. 57, 59

[11] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit.* Pearson Education, 2009. 2

[12] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. The formal specification language mcrl2. In *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007. 34, 36, 56

[13] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987. 3

[14] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 633–642, 2011. 2

[15] Daniel Jackson. *Software Abstractions: logic, language, and analysis.* MIT press, 2012. 34, 35

[16] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). 2

[17] Dale A Miller and Gopalan Nadathur. Higher-order logic programming. In *International Conference on Logic Programming*, pages 448–462. Springer, 1986. 39

[18] Rocco Nicola and Frits Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, pages 407–419. Springer Berlin Heidelberg, 1990. 3

[19] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society*, 39(2):25, 2006. 2

[20] Seyyed MA Shah, Kyriakos Anastasakis, and Behzad Bordbar. From uml to alloy and back again. In *International Conference on Model Driven Engineering Languages and Systems*, pages 158–171. Springer, 2009. 57

[21] Marten Sijtema, Axel Belinfante, MIA Stoelinga, and Lawrence Marinelli. Experiences with formal engineering: Model-based specification, implementation and testing of a software bus at neopost. *Science of computer programming*, 80:188–209, 2014. 3, 56, 59

[22] Terje Sivertsen. State-based and transition-based specifications, Feb 2001. 3

[23] Ulyana Tikhonova. Reusable specification templates for defining dynamic semantics of DSLs. *Software & Systems Modeling*, 18(1):691–720, March 2017. 2

[24] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008. 3, 43, 56

[25] Axel van Lamsweerde. Formal specification. In *Proceedings of the conference on The future of Software engineering - ICSE.* ACM Press, 2000. 3

[26] Jos B Warmer and Anneke G Kleppe. *The object constraint language: getting your models ready for MDA.* Addison-Wesley Professional, 2003. 7

[27] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications.

In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999. 34, 36

Appendix A

DMDSL specification in Alloy

A.1 Create

```
1  pred successful_create[s: State, s': State, e: Entity, targets: set Link]
2  {
3      e.constructability = True
4      #(instancesOfType[s.instances, e]) < e.multiplicity.maximum
5      targets.link_type = outgoing_associations[e]
6
7      some new : Instance {
8          new.type = e
9          new not in s.instances
10
11         all p: targets | p.link_source = new and p.link_target in s.instances
12
13         // validate association target multiplicity
14         all A: outgoing_associations[e] {
15
16             // validate target multiplicity
17             let num_links_at_source = outgoingLinksOfType[targets, new, A]
18             {
19                 #(num_links_at_source) >= A.targetProperty.multiplicity.
                       minimum
20                 #(num_links_at_source) <= A.targetProperty.multiplicity.
                       maximum
21             }
22
23             // validate source multiplicity for all target instances
24             all target_instance: instancesOfType[s.instances, A.target]
25             {
26                 let target_existing_links = #incomingLinksOfType[s.links, A,
                       target_instance]
27                 {
28                     let target_new_links = #incomingLinksOfType[targets, A,
                           target_instance]
29                     {
30                         plus[#target_existing_links, #target_new_links] <= A.
                               sourceProperty.multiplicity.maximum
31                     }
32                 }
```

```
            }
        }
        // define the contents of next state
        s'.instances = s.instances + new
        s'.repositories = s.repositories
        s'.links = s.links + targets
    }
```

Listing A.1: The Create transition as a predicate in Alloy

A.2 Update

```
pred successful_update[s: State, s': State, i_to_update: Instance, new_links:
    set Link] {
    i_to_update in s.repositories
    i_to_update.type.mutability = True

    one new_links.link_type and new_links.link_type in outgoing_associations[
        i_to_update.type]

    all p: new_links | p.link_source = i_to_update and p.link_target in s.
        repositories

    // ensure association target multiplicity will hold
    #(new_links) >= new_links.link_type.targetProperty.multiplicity.minimum
    #(new_links) <= new_links.link_type.targetProperty.multiplicity.maximum

    // ensure association source multiplicity will hold
    all T: new_links.link_target
    {
        let new_links_at_target =
            plus[ #(incomingLinksOfType[s.links, new_links.link_type, T]),
                #(incomingLinksOfType[new_links, new_links.link_type, T])]
        {
            #(new_links_at_target) <= new_links.link_type.sourceProperty.
                multiplicity.maximum
        }
    }

    s'.links = s.links
        - outgoingLinksOfType[s.links, i_to_update, new_links.link_type]
        + new_links
    s'.instances = s.instances
    s'.repositories = s.repositories

}
```

Listing A.2: The Update transition as a predicate in Alloy

A.3 Delete

```
pred successful_delete[s: State, s': State, x: Instance ] {

    x in s.repositories and x.type.deleteability = True

    #(instancesOfType[s.repositories, x.type]) > x.type.multiplicity.minimum

    all instance_to_delete: markedForDeletion[s, x] {
```

```
8           instance_to_delete.type.deleteability = True
9       }
10
11      all del_links: toBeDeletedLinks[s, x] {
12          del_links.link_source.type.deleteability = True or del_links.
                link_source.type.mutability = True
13      }
14
15      all e: Entity {
16          #instancesOfType[{s.instances - markedForDeletion[s, x]}, e] >= e.
                multiplicity.minimum
17      }
18      all a: Association {
19          all src : instancesOfType[s.instances - markedForDeletion[s, x], a.
                source]
20          {
21              let rem_links_src = outgoingLinksOfType[s.links - toBeDeletedLinks
                    [s, x], src, a]
22              {
23                  #(rem_links_src) >= a.targetProperty.multiplicity.minimum
24              }
25          }
26      }
27
28
29      // actions
30      s'.links = s.links - toBeDeletedLinks[s, x]
31      s'.repositories = s.repositories - markedForDeletion[s, x]
32      s'.instances = s.instances - markedForDeletion[s, x]
```

Listing A.3: The Delete transition as a predicate in Alloy