RWTH Aachen

Department of Computer Science
Technical Report

CD2Alloy: A Translation of Class Diagrams to Alloy

and Back from Alloy Instances to Object Diagrams

Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2017-06

RWTH Aachen · Department of Computer Science · July 2017

RWTH Aachen University
Software Engineering Group

CD2Alloy: A Translation of Class Diagrams to Alloy

and Back from Alloy Instances to Object Diagrams

Oliver Kautz
Shahar Maoz
Jan Oliver Ringert
Bernhard Rumpe

Abstract

This report presents a translation from UML class diagrams [OMG15, Rum16] to Alloy modules [Jac06] and a translation from Alloy instances back to UML object diagrams. An overview of the translation was first presented in [MRR11a] and applied in [MRR11b] to semantic differencing of class diagrams. It supports an extended list of CD language features, including, e.g., directed associations, composite aggregations, interfaces, multiple inheritance, and enumerations. The translation thus supports essential features of many real-world CDs, UML and EMF metamodels, practically not analyzable before. An important feature of the translation is the ability to analyze multiple class diagrams within one Alloy module, which is not possible with previous translations. This document defines the translations by translation rules that operate on the abstract syntax of a class diagram language and produce concrete syntax of the Alloy language. We give examples showing class diagrams and complete representations in Alloy as well as an Alloy instance and its object diagram representation.

Contents

Chapter 1

Introduction

Analyzing models of one modeling language can often be done using a semantics preserving translation to another language, and a reversed translation, back from the analysis results to the domain of the first language. Class diagrams (CDs) are widely used for modeling the structure of object-oriented systems and are the most popular sub-language of the Unified Modeling Language (UML) [OMG15] standard. The syntax of CDs includes classes and the various relationships between them such as associations and generalizations. The semantics of CDs is given in terms of object models, consisting of sets of objects and the relationships between these objects. Many authors have suggested different analyses problems, solutions, and related tools for CDs (e.g., [ABGR10, CCR07, GBR07]).

A class diagram specifies a model of an object-oriented system structure. Our approach relies on analyzed CDs using a translation to Alloy[1], a textual modeling language based on relational first-order logic [Jac06]. An Alloy module resulting from such a translation can be analyzed using a SAT solver. An analysis result, which is an instance of the module, if there is any, can be translated back to the UML domain, and be presented as an object diagram (OD). Existing translations of CDs to Alloy [ABGR07a, ABGR10, MGB04, SAB09] are limited to basic analyses of a single CD and lack automation of translating instances back to ODs. Moreover, the translations miss support for several CD language features as, for instance, multiple inheritance and interface implementation. The main reason for this is that these CD features do not have immediate counterparts in Alloy. In other words, the translations realize shallow embedding strategies.

This technical report presents CD2Alloy, a comprehensive translation of CDs to Alloy, which is based on a deeper embedding strategy. Rather than mapping each CD construct to a semantically equivalent Alloy construct, the translation presented in this report explicitly encodes the semantics of CD constructs in Alloy. Class inheritance, for instance, is not mapped to its Alloy's counterpart — the `extends` keyword. Instead, it is defined using several of Alloy's language constructs — facts, functions, and predicates. The semantics of the generated constructs then reflects the semantics of class inheritance in CDs. Earlier work [MRR11a] has introduced the basic features of CD2Alloy and presented the general idea of the translation without going into details. This report presents the translation from CDs to Alloy modules and back from Alloy instances to ODs in full detail.

The alternative translation has several advantages. First, it allows us to support more CD language features: in particular features that do not have direct counterparts in Alloy, such

[1] http://alloy.mit.edu/ accessed 2017-06-01

as multiple inheritance and interface implementation. Second, significantly, it allows to solve several analysis problems that go beyond the basic satisfiability check and instance generation tasks of a single CD, e.g., the analysis of the intersection of two CDs (i.e., generating common object models), the comparison of two CDs (checking if one is a refinement of the other), etc. These would have been very difficult, if not impossible, to support using existing translations from the literature. One of the strengths of CD2Alloy is the capability of producing witnesses for analysis results. Witnesses are presented in form of object diagrams, describing object models in the semantics of the class diagrams involved in the analysis. Producing witnesses is important because it provides correctness proofs. Browsing multiple witnesses supports human comprehension of the analysis results.

Technically, as concrete languages we use the CD and object diagram sublanguages of the UML/P [Sch12, Rum16], a conceptually refined and simplified variant of the UML designed for low-level design and implementation. Our semantics of CDs and ODs are based on [BCGR09, CGR08] and are given in terms of object models, i.e., sets of objects and relationships between these objects. The translation takes one or more CDs as input and outputs an Alloy module. The Alloy module can then be analyzed with the Alloy Analyzer. Finally, using another translation, instances of the Alloy module found by the SAT solver connected to the Alloy Analyzer are translated back to ODs. The transformations are presented in Chapter 3.

This report is structured as follows: Chapter 2 provides a brief summary on the UML/P CD language used by the translation and a short overview of Alloy. Chapter 3 describes the CD2Alloy translation from CDs to Alloy modules and illustrates the translation by example of a single input CD. Afterwards, Chapter 4 gives an example for the translation when given multiple CDs as input, before Chapter 5 describes the translation from Alloy instances back to ODs. Chapter 6 overviews related analysis approaches based on other translations. In the end Chapter 7 reflects the translation and concludes.

Chapter 2

Preliminaries

The translation takes as input a set of UML/P class diagrams [Rum16, Sch12] and outputs an Alloy module. The UML/P [Rum16] is a conceptually refined and simplified variant of the UML designed for low-level design and implementation. Alloy is a textual modeling language based on relational first-order logic. Alloy modules can be analyzed with the Alloy analyzer, a fully automated constraint solver. The analyzer can check the validity of user defined predicates and is capable of finding counterexamples during analysis for a user defined finite scope. Alloy instances computed by the Alloy analyzer for modules produced by the translation can be translated back to object diagrams. The object diagrams represent object models in the semantics of the input CDs calculated for an analysis problem specified for the Alloy module generated from the input CDs.

Section 2.1 presents the class diagram language used as input for the translation, before Section 2.2 shows an example class diagram that is used as a running example. In the end of this chapter, Section 2.3 gives a brief overview of Alloy.

2.1 Grammar of UML/P CDs

MontiCore [KRV10] is a language workbench for the development of compositional modeling languages. It supports the definition and generation of all artifacts relevant for language processing for a modeling language specified by a grammar in an enriched EBNF format. The generated artifacts include, inter alia, the abstract and concrete syntax of a language and a parser for models conforming to the language.

Listing 2.1 shows the relevant parts of the class diagram MontiCore grammar describing the language of the class diagrams used as input for the CD2Alloy translation. The grammar is adapted from [Sch12] and defines a subset of the UML/P [Rum16] class diagram language. The complete syntax of the UML/P CD language is defined in [Sch12] using MontiCore grammars and context conditions for describing well-formedness rules. The grammar depicted in Listing 2.1 extends the grammar `mc.types.Types` and thus inherits, inter alia, the production rules `Type` for stating standard and primitive types and `ReferenceType` for references to type names of classes or interfaces. A detailed description of the grammar `Types` is given in [Sch12].

MCG

```
1 grammar CD extends mc.types.Types {
2   CDDefinition = "classdiagram" Name
3           "{" ( Class | Interface | Enum | Association )* "}";
4   Class = Stereotype? Modifier? "class" Name
5         ( "extends" superclasses:ReferenceType
6                ("," superclasses:ReferenceType)* )?
7         ( "implements" interfaces:ReferenceType
8                   ("," interfaces:ReferenceType)* )?
9         ( ";" | "{" Attribute* "}" );
10  Interface = "interface" Name ("extends" interfaces:ReferenceType
11                                    ("," interfaces:ReferenceType)* )?";";
12  Enum = "enum" Name ( ";" | "{" EnumConstant ("," EnumConstant)* ";" "}");
13  EnumConstant = Name;
14  Attribute = Type Name";";
15  Association = ( ["association"] | ["composition"] )
16        leftCardinality:Cardinality?
17        leftReferenceName:QualifiedName
18        ("(" leftRole:Name ")" )?
19        ( leftToRight:  ["->"]  | rightToLeft:["<-"] |
20          bidirectional:["<->"] | simple:     ["--"] )
21        ("(" rightRole:Name ")")?
22        rightReferenceName:QualifiedName
23        rightCardinality:Cardinality?
24  Cardinality = "[" ( many:["*"] | lower:IntLiteral ( ".."
25            ( upper:IntLiteral | noUpperLimit:["*"]) )? ) "]";
26  Modifier = ( Abstract:["abstract"] )*;
27 }
```

Listing 2.1: Extract of a MontiCore grammar for UML/P class diagrams.

The definition of a class diagrams starts with the keyword `classdiagram` followed by the diagram's name and body. The body is enclosed by square brackets (l. 3). It contains arbitrarily many classes, interfaces, enums, and associations (l. 3). Classes are introduced with the keyword `class`, optionally prefixed with a stereotype and a modifier (l. 4). The keyword must be followed by the name of the class. The name is optionally followed by the keyword `extends` and a list of references to classes the class extends (ll. 5-6) or by the keyword `implements` and a list of interfaces the class implements (ll.7-8). Afterwards, a class definition optionally has a body enclosed by square brackets (l. 9). The body consists of arbitrarily many attributes (l. 9). Each attribute consists of a type and a name (l. 14). Stereotypes are represented as lists of stereotype identifiers enclosed by angle brackets (”≪” and ”≫”) and are part of the `Types` grammar. The translation currently supports the modifier `abstract` (l. 26) and the stereotype identifier `singleton`. Interfaces are introduced with the keyword `interface` followed by the interface's name (l. 10). Each interface can optionally extend further interfaces (ll. 10-11). The definition of an enumeration type starts with the keyword `enum` followed by the enumeration's name (l. 12). The name is optionally followed by a body that is enclosed by square brackets and defines the fields of the enumeration type (l. 12). The grammar supports declaring regular associations, introduced with the keyword `association`, and compositions, introduced with the keyword `composition` (l. 15). Each association defines the classes it associates (l. 17 and l. 22) and is either unidirectional (denoted by `->` or `<-`, l. 19), undirected (denoted by `--`, l. 20), or bidirectional (denoted by `<->`, l. 20). Association ends are optionally labeled with cardinalities (l. 16 and l. 23) and role names (l. 18 and l. 21). If a role name on a side is omitted, it is inferred as the name of the class associated on the same side starting with a lower case letter. Cardinalities are of the form `*`, `n`, `n..*`, or `n..m` (ll. 26-27) where `n` and `m` are integers with `n` < `m`.

2.2 Example CD

Figure 2.1 shows an example of a UML/P class diagram in graphical notation. Listing 2.2 shows the same class diagram in textual notation. The class diagram contains two enumerations, eight classes of which one is abstract, one interface, four regular associations, one composition, three class inheritance relations, and one implements relation.

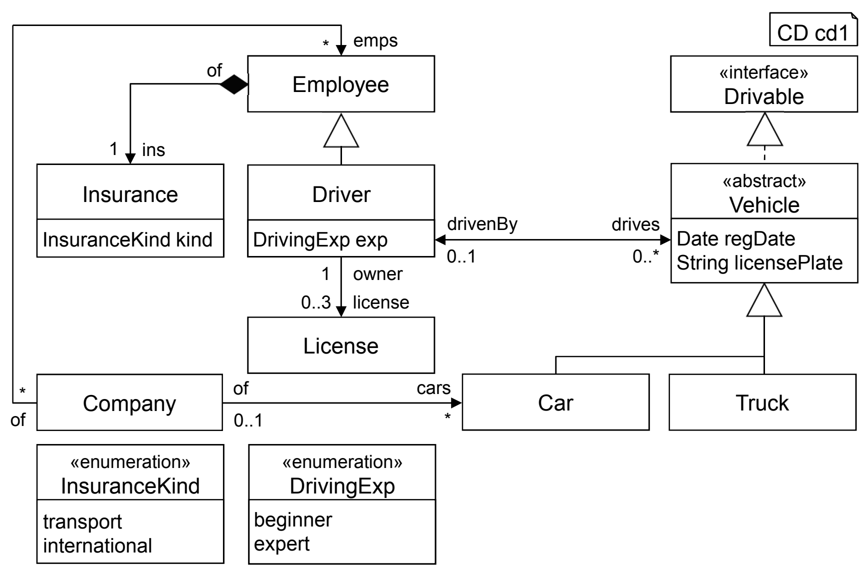

Figure 2.1: The example class diagram cd_1 consists of classes with attributes, enumerations with enumeration values, associations with multiplicities, inheritance relations between classes, and an implements relations between a class and an interface.

Each employee has exactly one insurance and can work in arbitrarily many companies. Insurances have a `kind` of enumeration type `InsuranceKind`, which has the possible enumeration values `transport` and `international`. Companies can own arbitrarily many cars. Drivers are special employees that have a driving experience with the possible enumeration values `beginner` or `expert`. Drivers can have up to three licenses and drive an arbitrary number of vehicles. Each vehicle is driven by up to one driver. The class `Vehicle` is an abstract class that implements the interface `Drivable`. There are two concrete types of vehicles: `Car` and `Truck`.

2.3 Brief Overview of Alloy

Alloy[1] is a modeling language based on relational first-order logic [Jac06]. The models written in Alloy are called modules. An Alloy module consists of signature declarations,

[1]http://alloy.mit.edu/ accessed 2016-11-07

CD cd1

```
classdiagram cd1 {
  enum InsuranceKind {transport, international;}
  enum DrivingExp {expert, beginner;}
  class Employee;
  class Driver extends Employee {
    DrivingExp exp;
  }
  interface Driveable;
  abstract class Vehicle implements Driveable {
    Date regDate;
    String licensePlate;
  }
  class Car extends Vehicle;
  class Truck extends Vehicle;
  class Company {}
  class License {}
  class Insurance {
    InsuranceKind kind;
  }
  association [1] Driver (drivenBy) <-> (drives) Car [0..*];
  association [0..1] Company (of) -> (cars) Car [*];
  association [*] Employee (emps) <- (of) Company [*];
  composition [1] Employee (of) -> (ins) Insurance [1];
  association [1] Driver (owner) -- (license) License [0..3];
}
```

Listing 2.2: The class diagram cd_1 (`CD2AlloyExample`) given in UML/P CD concrete syntax.

fields, facts and predicates. Each signature denotes a set of atoms, which are the basic entities in Alloy. Relations between two or more signatures are represented using fields and are interpreted as sets of tuples of atoms. Facts are statements that define constraints on the elements of the model. Predicates are parametrized constraints. A predicate can be included in other predicates or facts.

Alloy modules can be analyzed using the Alloy Analyzer, a fully automated constraint solver. This is done by a translation of the module into a Boolean expression. The expression is analyzed by SAT solvers embedded within the Analyzer. The analysis is based on an exhaustive search for instances of the module, bounded by a user-specified scope. The scope limits the number of atoms for each signature in an instance of the system that the solver analyzes. The Analyzer can check for the validity of user-specified assertions. If an instance that violates the assertion is found within the given scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope. Used in the opposite way, the Analyzer can search for instances of user-specified predicates. If the predicate is satisfiable within the given scope, the Analyzer will find an instance that proves it. However, if the analyzer does not find an instance, the predicate may be satisfiable in a larger scope. For a complete and detailed account of Alloy we refer to [Jac06].

Chapter 3

Translation of Class Diagrams to Alloy Modules

This chapter defines the rules for translating class diagrams to Alloy modules. The input of the translation is a set of UML/P class diagrams. The output is an Alloy module containing a predicate for each class diagram. Each of such predicates expresses the semantics of its corresponding CD in terms of Alloy instances representing the object models in the semantics of the CD.

Section 3.1 describes the abstract syntax of the class diagram language used as input for the translation. Afterwards, Section 3.2 presents the translation rules for transforming CDs to Alloy modules. The rules are illustrated by example of the class diagram described in Section 2.2. Appendix A shows the complete result from applying the CD2Alloy translation to the class diagram given in Section 2.2.

3.1 Abstract Syntax of Class Diagrams

The translation rules operate on the abstract syntax of class diagrams as defined by the simplified MontiCore grammar described in Section 2.1 and depicted in Listing 2.1.

The grammar format provided by MontiCore [KRV10] is an extended CFG format enhanced with the possibility for the specification of concrete and abstract syntax. MontiCore derives an abstract syntax tree (AST) from each grammar. An AST is a data structure for representing the abstract syntax of a language defined by a MontiCore grammar. Additionally, MontiCore derives a class diagram representing the AST of the language defined by a grammar. Code generators translate the class diagram into Java code. Parsers derived from MontiCore grammars instantiate these classes and thus create AST instances representing the abstract syntax of models. The descriptions of the translations presented in this report abstract from implementation details and deal with AST instances on a conceptual level. The translation rules handle AST instances in a similar manner as one operates on mathematical structures.

The AST of a language can be represented by a class diagram. The class diagram consists of a class for each production rule of the corresponding grammar. The name of the class is defined by the name of the non-terminal introduced by the left-hand side of the production rule. Each class in the class diagram represents an AST node. The associations between the classes are defined by the right-hand sides of the production rules. Each production rule has a name and a body. The name introduces the name of the non-terminal defined by the rule. Elements referenced by the the body of production rules can be explicitly named. If the name of an element is omitted, the name is derived from the name of the referenced terminal or non-terminal. Repetition of an element on the right hand of a production rule is denoted by the `*` symbol. Optional elements are followed by the symbol `?`. The special non-terminal `Name` induces a universe of names. Referencing the `Name` non-terminal on the right-hand side of a production rule introduces a field in the corresponding AST node class. The field ranges over the universe induced by `Name`. Each name is required to have an unique String representation. If a production rule p references a non-terminal n different from `Name`, the AST data structure contains an association between the classes introduced for p and for the production corresponding to n. The association is navigable from the class for p to the class for n. The role name given to the class corresponding to n is given by the name of the element referencing n. The association's cardinality on the side of the class introduced for n is the same as specified in the grammar: (1) If the referenced non-terminal is marked as optional, the classes are related in a one to at most one relationship. (2) If the referenced non-terminal is part of a repetition, the classes are related in a one to many relationship. (3) Otherwise, the classes are related in a one to one relationship. Keywords in square brackets on the right hand side of a production rule are added to the corresponding AST class as boolean attributes. A detailed description of the derivation of the abstract syntax of MontiCore languages is given in [KRV10].

The root of any class diagram AST is a `CDDefinition` node. For the purpose of navigating over an AST instance, the transformations presented in this report use the well known dot notation. For instance, a class diagram represented by a `CDDefinition` node cd contains the set of classes $cd.class$, the associations $cd.association$, the enumeration types $cd.enum$, and the interfaces $cd.interface$. As a second example, the expression $\{a.name \mid \exists c \in cd.class : a \in c.attribute\}$ describes the set of all names of all attributes of all classes occurring in the CD cd. In the remainder of this report, the reflexive transitive closure of the `superclasses` and `interfaces` attributes of a class $c \in cd.class$ are denoted by $c.superclasses^*$ and $c.interfaces^*$, respectively. For a class c, the expression $c.superclasses^*.interfaces^*$ denotes the set of all interfaces implemented by any class contained in the set $c.superclasses^*$ and is defined by the equation $c.superclasses^*.interfaces^* = \bigcup_{sc \in .superclasses^*} sc.interfaces^*$. The translation rules presented in this report often require checking equality of names. For notational convenience the translation rules use the following abbreviations for this purpose: Given a name n and an AST node ast, we denote by $n = ast$ the expression that evaluates to true if, and only if, the value of the `name` attribute of the AST node ast is equal to the name n. For instance, given a name n and a `Type` node t, the expression $n = t$ evaluates to true iff the name of the type represented by t is equal to the name n. Analogously, given two AST nodes ast and ast', the expression $ast = ast'$ evaluates to true iff $ast.name = ast'.name$. Given a `Class` node c and an `Interface` node i, for instance, the expression $c = i$ evaluates to true iff $c.name = i.name$.

3.2 The CD2Alloy Translation Rules

The CD2Alloy translation takes as input a set of class diagrams CD. It outputs a single Alloy module containing a predicate for each class diagram $cd \in CD$. Each predicate expresses the semantics of the corresponding CD in terms of Alloy instances that represent the object models in the semantics of the CD. The translation is divided into four stages. The first stage (cf. Section 3.2.1) is the generic part of the CD2Alloy translation producing signatures, facts, and predicates common to all CD2Alloy translations. The second stage (cf. Section 3.2.2) creates signatures for the representation of CDs in CD2Alloy. This stage produces signatures common to all CDs used as input for the translation. The third stage (cf. Section 3.3) creates functions for expressing subclassing, interfaces, composition, and enumeration types. Each of the functions is specific to exactly one $cd \in CD$. The fourth and last stage (cf. Section 3.4 and Section 3.5) creates an Alloy predicate for each $cd \in CD$ that expresses the semantics of the CD in terms of valid Alloy instances representing object models in the semantics of the CD *cd*.

The translation starts with the translation rule depicted in Figure 3.1. The rule is defined using the formal notation for translation rules as defined in Appendix B of [Rin14]. The translation rules consist of expressions in the concrete syntax of the target language Alloy as well as control structures and directives that operate over the abstract syntax of class diagrams as described in Section 3.1. The translation rule syntax and the effect of applying transformations is described in detail in [Rin14], Appendix B.

The key idea of the CD2Alloy translation is the definition of CD semantics using custom predicates instead of using the built-in concepts of the Alloy language. This has multiple advantages. First, the CD2Alloy translation allows to express features that cannot be expressed by direct mappings of CD constructs to Alloy constructs. A detailed discussion of the differences between the semantics of CDs and the Alloy language is given in [ABGR10]. Alloy's `extends` keyword, for instance, cannot handle multiple inheritance or interface implementation. Second, the CD2Alloy translation allows the analysis of multiple CDs using Alloy. The semantics of a CD is not fixed by Alloy constructs, such as fields of signatures, but defined in a custom predicate. Thus, analysis tasks involving multiple CDs can be written as an expression over multiple predicates and evaluated automatically by the Alloy Analyzer.

3.2.1 The Generic Part

The generic part of the CD2Alloy translation is common to all generated modules, independent of the input CDs. It consists of a set of parametrized auxiliary predicates for expressing the semantics of CDs and a fixed core of abstract Alloy signatures for representing class instances (objects), field names, field values, and enumeration type values.

Listing 3.1 shows the abstract signature `Obj` (l. 1) that is the parent of all signatures representing classes in the module. The `get` Alloy field of the `Obj` signature relates `Obj` and `FName` atoms to atoms of the `Obj`, `Val`, and `EnumVal` signatures. The abstract signature `FName` (l. 2) is used to represent association role names and attribute names for all classes in the CDs. The abstract signature `Val` (l. 3) represents all predefined and unknown types, i.e., primitive types and other types that are not defined as classes in a CD. Values of enumeration types are represented using the signature `EnumVal` (l. 4). All the Alloy signatures above are abstract and thus have no immediate instances. The signatures are extended by signatures representing elements from CDs in the CD specific parts of the translation.

Listing 3.2 and Listing 3.3 show the generic, parametrized predicates responsible for specifying the relations between objects and fields. The predicate `ObjAttrib` (Listing 3.2,

Translation Rule

Translation rule with a set CD of class diagrams as parameter:

```
module {cd.name}_{cd∈CD}
// generic signatures and classes   ⓘ see Section 3.2.1
// signatures common to all CDs   ⓘ see Section 3.2.2
executeRule(U1 CD)   ⓘ see Figure 3.2 for body of rule U1
executeRule(U2 CD)   ⓘ see Figure 3.3 for body of rule U2
executeRule(U3 CD)   ⓘ see Figure 3.4 for body of rule U3
executeRule(U4 CD)   ⓘ see Figure 3.5 for body of rule U4
∀cd ∈ CD :
  // functions specific to CD cd.name   ⓘ see Section 3.3
  executeRule(F1 cd)   ⓘ see Figure 3.6 for body of rule F1
  executeRule(F2 cd)   ⓘ see Figure 3.7 for body of rule F2
  executeRule(F3 cd)   ⓘ see Figure 3.8 for body of rule F3
  executeRule(F4 cd)   ⓘ see Figure 3.9 for body of rule F4
∀cd ∈ CD :
  // semantics predicate cd.name   ⓘ see Section 3.4 and Section 3.5
  pred cd.name {
    // classes and attributes in cd.name   ⓘ see Section 3.4
    executeRule(P1 cd)   ⓘ see Figure 3.11 for body of rule P1
    executeRule(P2 cd)   ⓘ see Figure 3.12 for body of rule P2
    executeRule(P3 cd)   ⓘ see Figure 3.13 for body of rule P3
    executeRule(P4 cd)   ⓘ see Figure 3.14 for body of rule P4
    // associations in cd.name   ⓘ see Section 3.5
    executeRule(A1 cd)   ⓘ see Figure 3.15 for body of rule A1
    executeRule(A2 cd)   ⓘ see Figure 3.16 for body of rule A2
    executeRule(A3 cd)   ⓘ see Figure 3.17 for body of rule A3
    executeRule(A4 cd)   ⓘ see Figure 3.18 for body of rule A4
    executeRule(A5 cd)   ⓘ see Figure 3.19 for body of rule A5
    executeRule(A6 cd)   ⓘ see Figure 3.20 for body of rule A6
  }
```

Figure 3.1: Overview of the translation of a set of class diagrams CD into an Alloy module.

Alloy

```
abstract sig Obj { get: FName -> {Obj + Val + EnumVal} }
abstract sig FName {}
abstract sig Val {}
abstract sig EnumVal {}
```

Listing 3.1: The abstract signatures `FName`, `Obj`, `Val`, and `EnumVal`.

Alloy

```
1 pred ObjAttrib[objs: set Obj, fName: one FName,
2                fType: set {Obj + Val + EnumVal}] {
3   objs.get[fName] in fType
4   all o: objs| one o.get[fName] }
5
6 pred ObjFNames[objs:  set Obj, fNames:set  FName] {
7   no objs.get[FName - fNames] }
8
9 pred BidiAssoc[left:  set Obj, lFName:one FName,
10               right: set Obj, rFName:one FName] {
11   all l: left | all r: l.get[lFName] | l in r.get[rFName]
12   all r: right | all l: r.get[rFName] | r in l.get[lFName] }
13
14 pred Composition[compos: Obj->Obj, right: set Obj] {
15   all r: right | lone compos.r }
16
17 fun rel[wholes: set Obj, fn: FName]  : Obj->Obj {
18   {o1:Obj,o2:Obj|o1->fn->o2 in wholes <: get} }
```

Listing 3.2: Parametrized predicates for specifying the relations between objects and fields of objects.

ll. 1-4) limits the tuples in the `objs.get[fName]` relation to the correct type of the field `fName`, which is given by the set `fType`. The predicate ensures exactly one object, value, or enumeration value is related to each object represented by an atom in the set `objs` and to the field name represented by the atom `fName`. The predicate `ObjFNames` (Listing 3.2, ll. 6-7) is used to ensure objects do not have field names other than the ones stated in the CD. It requires the `get` relation does not relate any atom in the set `objs` with any atom not contained in the set `fNames` to any object or value. If the set `objs` consists of all atoms representing all instances of a class and the set `fNames` contains all atoms representing all attribute names of the class as well as all all role names given to classes associated with the class, the predicate ensures no instance of the class has a field that is not stated in the CD. The predicate `BidiAssoc` (Listing 3.2, ll. 9-12) is used to ensure each object having a link to another object via a bidirectional association can also be referenced by the other object via a link representing an instance of the same association. The association's role names are represented by the atoms given by the parameters `lFName` and `rFname`. The predicate requires that all partners on the left ends of links corresponding to the association have links back to the partners on the right ends of the association and vice versa. There are different interpretations of composition semantics in the literature. For instance, [GR99] requires the part to be existentially dependent from the aggregate and requires a strong form of forbidding sharing, i.e., a part object can only exist if it is connected to a whole object and a part object cannot be shared by two different whole objects. In contrast, in the UML [OMG15] and in the UML/P [Rum16], it is only required that each part object is related to at most one whole object. This report considers the semantics specified for the UML/P [Rum16]. The predicate `Composition` (Listing 3.2, ll. 14-15) is used to ensure that each part object is connected to at most one whole. It states the `compos` relation relates each atom contained in the set `right` at most once, i.e., there is at most one `(l,r)` ∈ `compos` such that `r = right`. If the set `right` contains all atoms representing all objects of a specific class and the `compos` relation contains all whole/part tuples such that the second component of each tuple contains a part object that is an instance of the specific class and the first component contains a

Alloy

```
1 pred ObjUAttrib[objs: set Obj, fName:one FName, fType:set Obj, up: Int] {
2   objs.get[fName] in fType
3   all o: objs| (#o.get[fName] =< up) }
4
5 pred ObjLAttrib[objs: set Obj, fName: one FName, fType: set Obj, low: Int] {
6   objs.get[fName] in fType
7   all o: objs | (#o.get[fName] >= low) }
8
9 pred ObjLUAttrib[objs:set Obj, fName:one FName, fType:set Obj,
10                  low: Int,     up: Int] {
11   ObjLAttrib[objs, fName, fType, low]
12   ObjUAttrib[objs, fName, fType, up] }
```

Listing 3.3: Predicates used to specify cardinality constraints for navigable association ends and for association ends of undirected associations.

whole object from which navigation is possible to an object of the given class, the predicate ensures each part object of the specific class can be referenced by at most one whole. The function `rel` (Listing 3.2, ll. 17-18) takes as input a set of `Obj` atoms `wholes` and a `FName` atom `fn`. It returns a relation between `Obj` atoms. The relation consists of all tuples of `Obj` atoms (`o1`,`o2`) that are related with the atom `fn` by the `get` relation and where the first component is a member of the set `wholes`. Intuitively, the function returns all pairs of objects where the first component of the tuple is a member of the objects represented by the set `wholes` and the second component of each pair can be referenced by the first component by using the field name represented by the atom `fn`.

Listing 3.3 shows the generic predicates used to specify cardinality constraints for navigable association ends and for association ends of undirected associations. The predicate `ObjUAttrib` (ll. 1-3) defines the set of possible partners of links and provides an upper bound for the number of objects related by the `get` relation for a specific role name and a specific object. First, it requires all atoms related to any atom in the set `objs` and the field name `fName` via the `get` relation are members of the set `fType`. Second, it requires the get relation relates each atom in the set `objs` with the atom `fName` to at most `up` many other atoms. If the set `objs` contains all atoms representing all instances of a class, the parameter `fName` represents a field name of the class, the set `fType` contains all atoms representing all instances of the field's type, and the parameter `up` is equal to the field's corresponding multiplicity, the predicate ensures links for the field `fName` only relate the given objects to other objects of the field's type and the number of links is bounded by the corresponding association's upper cardinality. The predicate `ObjLAttrib` (ll. 5-7) is defined analogously. It is used to specify lower bounds of associations. The two predicates `ObjLAttrib` and `ObjUAttrib` are both used by the predicate `ObjLUAttrib` (ll. 9-12) to define association ends with lower and upper multiplicities.

The predicates depicted in Listing 3.4 are used to specify cardinality constraints for non-navigable association ends. Given an object represented by an element of the set `objs`, the predicate `ObjL` (ll. 1-2) provides a lower bound `low` for the number of objects, represented by the atoms contained in the set `fType`, from which navigation must be possible to the given object via the role name represented by the atom `fName`. The predicate `ObjU` (ll. 4-5) is defined analogously for specifying upper bounds and the predicate `ObjLU` (ll. 7-10) can be used to specify both, lower and upper bounds.

Alloy

```
pred ObjL[objs: set Obj, fName:one FName, fType: set Obj, low: Int] {
  all r: objs | # { l: fType | r in l.get[fName]} >= low }

pred ObjU[objs: set Obj, fName:one FName, fType: set Obj, up: Int] {
  all r: objs | # { l: fType | r in l.get[fName]} =< up }

pred ObjLU[objs: set Obj, fName:one FName, fType: set Obj,
           low: Int,       up: Int] {
  ObjL[objs, fName, fType, low]
  ObjU[objs, fName, fType, up] }
```

Listing 3.4: Parametrized predicates used to specify cardinality constraints for non-navigable association ends.

Translation Rule

Translation rule with a set CD of class diagrams as parameter:

U1 $\forall c \in \bigcup_{cd \in CD} cd.class$:
`sig` $c.name$ `extends Obj {}`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
sig Vehicle extends Obj {}
sig Company extends Obj {}
sig Employee extends Obj {}
sig Car extends Obj {}
sig Insurance extends Obj {}
sig License extends Obj {}
sig Driver extends Obj {}
sig Truck extends Obj {}
```

Figure 3.2: Rule U1 generates a signature extending the signature `Obj` for each class of any class diagram in the set CD.

3.2.2 Rules U1 to U4: Classes, Field Names, Types, and Enums

The rules U1 to U4 generate signatures for CD elements collected from the union of the elements of all input CDs. The signatures for classes, field names, primitive types, and enumerations are declared for all CDs. The rules described in Section 3.3, Section 3.4, and Section 3.5 generate functions and predicates for constraining the relations between atoms of the signatures generated from rules U1 to U4 for each individual input CD.

Rule U1 shown in Figure 3.2 creates an Alloy signature extending the signature `Obj` for every class in the union of classes from all class diagrams. Atoms of these signatures represent objects in the object models in the semantics of the CDs. As interfaces can not be instantiated, the translation does not introduce signatures for interfaces. However, the translation creates signatures for abstract classes. The translation rule P3 (cf. Figure 3.13) introduced later assures no object model in the semantics of a CD contains instances of abstract classes. Translating the CD given in Listing 2.2 produces the results shown in the lower part of Figure 3.2.

Translation Rule

Translation rule with a set CD of class diagrams as parameter:

U2 $\forall n \in \{a.name \mid \exists c \in \bigcup_{cd \in CD} cd.class : a \in c.attribute\} \cup$
$\{a.leftRole, a.rightRole \mid a \in \bigcup_{cd \in CD} cd.association\}$:
`one sig` n `extends FName {}`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
one sig owner extends FName {}
one sig cars extends FName {}
one sig license extends FName {}
one sig licensePlate extends FName {}
one sig emps extends FName {}
one sig drives extends FName {}
one sig kind extends FName {}
one sig of extends FName {}
one sig regDate extends FName {}
one sig exp extends FName {}
one sig drivenBy extends FName {}
one sig ins extends FName {}
```

Figure 3.3: Rule U2 produces a singleton signature for each attribute and for each association role name occurring in any class diagram of the set CD.

The translation Rule U2 defined in Figure 3.3 introduces a singleton signature for every name used as field name in any class of any CD contained in the set CD. Therefore, the rule iterates over all attribute names of any class and over all role name of any association of any CD contained in the set CD. All the signatures extend the abstract signature `FName`. The signatures' atoms are therefore related by the `get` relation to atoms representing values of attributes and objects. The translation of the CD given in Listing 2.2 produces the results shown in the lower part of Figure 3.3.

The rule U3 depicted in Figure 3.4 creates a singleton signature extending the signature `Val` for every type of an attribute of a class defined in a CD that is not defined by a class or an interface in the CD. Each of these signatures represents an instance of a primitive or an unknown type. The constraint $\forall t \in cd.class \cup cd.interface : a.type \neq t.name$ ensures the type $a.type$ is not defined by any class or interface in the CD cd. If a type is defined by a class or by an interface in a class diagram cd' but not in a class diagram cd, the type is still treated as a primitive or unknown type in the CD cd. Since the range of unknown types is not known and the range of primitive types is not of importance for the analysis, the signatures are assigned the multiplicity `one`. This ensures all objects share a single symbolic instance of each primitive or unknown type. The lower part of Figure 3.4 shows the results from translating the class diagram depicted in Listing 2.2.

Translation Rule U4 shown in Figure 3.5 creates a signature for every enumeration value defined in any CD contained in the set CD. To ensure all objects share a single symbolic instance of each enumeration value, the signatures have the multiplicity `one`. The name of each signature includes the name of the corresponding enumeration type to distinguish same named values of different enumerations. The enumeration constant `Orange` of an enumeration named `Fruit`, for instance, has to be distinguishable from the enumeration

Translation Rule

Translation rule with a set CD of class diagrams as parameter:

U3 $\forall type \in \{a.type \mid \exists cd \in CD : \exists c \in cd.class : a \in c.attribute \wedge$
$\forall t \in cd.class \cup cd.interface : a.type \neq t.name\} :$

`one sig type_`*type* `extends Val {}`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
one sig type_Date extends Val {}
one sig type_String extends Val {}
```

Figure 3.4: Rule U3 creates a singleton signature for each primitive or unknown type used in any CD contained in the set CD.

Translation Rule

Translation rule with a set CD of class diagrams as parameter:

U4 $\forall e \in \bigcup_{cd \in CD} cd.enum : \forall v \in e.enumConstant :$

`one sig enum_`*e.name*`_`*v.name* `extends EnumVal {}`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
one sig enum_InsuranceKind_international extends EnumVal {}
one sig enum_DrivingExp_expert extends EnumVal {}
one sig enum_InsuranceKind_workAcc extends EnumVal {}
one sig enum_InsuranceKind_transport extends EnumVal {}
one sig enum_DrivingExp_beginner extends EnumVal {}
```

Figure 3.5: Rule U4 produces a signature for every enumeration value defined in any CD that is a member of the set CD.

value `Orange` of an enumeration named `Color`. Each signature is independent of the class diagram defining it to be able to consider two same named constants of two same named enumerations of two different CDs to be equal. Two `Color` constants `Orange` defined in two same named enumerations in two different class diagrams, for instance, need to be considered to be equivalent in the object models in the semantics of the CDs. The results from applying the translation rule to the class diagram shown in Listing 2.2 are depicted in the lower part of Figure 3.5.

3.3 Rules F1 to F4: Functions for Subclassing, Interfaces, Compositions, and Enums

As indicated in Figure 3.1, the rules F1 to F4 are executed for every class diagram $cd \in CD$. Each rule creates Alloy functions to access atoms representing values and objects in the semantics of the translated CDs.

Translation Rule

Translation rule with parameter $cd \in CD$:

F1 $\forall c \in cd.class:$

fun $c.name$SubsCD$cd.name$:set Obj {
let $subs = \{sub \in cd.Class \mid c \in sub.superclasses^*\}$ **in**
$\{\!|sub.name|\!\}^{+}_{sub \in subs}$
}

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
1 fun VehicleSubsCDcd1: set Obj { Vehicle + Car + Truck }
2 fun CompanySubsCDcd1: set Obj { Company }
3 fun EmployeeSubsCDcd1: set Obj { Employee + Driver }
4 fun CarSubsCDcd1: set Obj { Car }
5 fun InsuranceSubsCDcd1: set Obj { Insurance }
6 fun LicenseSubsCDcd1: set Obj { License }
7 fun DriverSubsCDcd1: set Obj { Driver }
8 fun TruckSubsCDcd1: set Obj { Truck }
```

Figure 3.6: Rule F1 produces a function for each class in the CD *cd* returning all atoms of all subclasses of the class.

The rule F1 defined in Figure 3.6 produces a function for each class in the CD that returns all atoms representing all subclass instances of the class. The functions support expressing subclassing. Since the functions are CD specific, it is possible to express a different inheritance hierarchy for each individual CD. The set defining predicate of the set *subs* requires *c* to be a member of the reflexive transitive closure of the superclass relation of the class *sub*. Therefore, the transition always produces a well formed Alloy function since the set *subs* is never empty because it always at least contains the class *c*. The function produced for the class `Car` and the class diagram given in Listing 2.2, which is shown in Figure 3.6 l. 4, for instance, returns the set of all `Car` atoms.

Translation rule F2 shown in Figure 3.7 introduces a function for every interface in the CD that returns all atoms representing instances of classes implementing the interface. Since interface implementation can be inherited, the rule also selects the classes having superclasses implementing the interface. The two transitive and reflexive closures of attributes *superclasses* and *interfaces* also capture super interfaces of interfaces implemented by superclasses. In contrast to the set *subs* defined in translation rule F1, the set *impls* described in translation rule F2 can be empty in case an interface is not implemented by any class. Thus the functions produced for interfaces that are not implemented by any class return `none`. For any other interface, the generated function returns the union of the atoms representing all instances of the classes implementing the interface. The translation of the class diagram shown in Listing 2.2 produces the results shown in the lower part of Figure 3.7.

The rule F3 defined in Figure 3.8 creates a function for each enumeration type in the CD that returns the atoms representing the enumeration type's possible values. The values are given by the signatures produced by translation rule U4 shown in Figure 3.5. The function

Translation Rule

Translation rule with parameter $cd \in CD$:

F2 $\forall i \in cd.interface:$

$$\underline{\texttt{fun }}\, i.name\underline{\texttt{SubsCD}}cd.name\ \underline{\texttt{:set Obj \{}}$$
$$\textbf{let}\ impls = \{c \in cd.class \mid i \in c.superclasses^{*}.interfaces^{*}\}\ \textbf{in}$$
$$\quad \textbf{if (} |impls| > 0 \textbf{) then } \{\!|c.name|\!\}^{\underline{+}}_{c \in impls}\ \textbf{else}\ \underline{\texttt{none}}$$
$$\underline{\texttt{\}}}$$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
fun DriveableSubsCDcd1: set Obj { Car + Truck + Vehicle }
```

Figure 3.7: Rule F2 produces a function for each interface in the CD *cd* returning all instances of classes implementing the interface.

Translation Rule

Translation rule with parameter $cd \in CD$:

F3 $\forall e \in cd.enum:$

$$\underline{\texttt{fun }}\, e.name\underline{\texttt{EnumCD}}cd.name\ \underline{\texttt{:set EnumVal \{}}$$
$$\quad \{\!|\underline{\texttt{enum\_}}e.name\underline{\texttt{\_}}enumVal.name|\!\}^{\underline{+}}_{enumVal \in e.enumConstant}$$
$$\underline{\texttt{\}}}$$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
fun DrivingExpEnumCDcd1: set EnumVal {
  enum_DrivingExp_expert + enum_DrivingExp_beginner }
fun InsuranceKindEnumCDcd1: set EnumVal  {
  enum_InsuranceKind_international +
  enum_InsuranceKind_transport }
```

Figure 3.8: Rule F3 creates a function for each enumeration type in the CD *cd* that returns the enumeration's possible values.

generated for a specific CD only returns the atoms representing enumeration values defined by the enumeration type defined in the given CD. It does not return atoms corresponding to enumeration values defined by same named enumeration types in other CDs that are not defined by the enumeration type of the given CD. The lower part of Figure 3.8 shows the result produced from translating the class diagram depicted in Listing 2.2.

The rule F4 shown in Figure 3.9 creates a function for every part participating in a composite whole-part-relation in the CD. It returns all linked whole/part instance pairs where the first component is an instance of a whole and the second component is an instance of the part. The bottom of Figure 3.9 shows the result produced by the translation given the class diagram depicted in Listing 2.2 as input. The class diagram contains

Translation Rule

Translation rule with parameter $cd \in CD$:

F4 $\forall part \in \{a.RightReferenceName \mid a \in cd.association \wedge a.isComposition\}$:

let $comps = \{comp \in cd.Association \mid comp.isComposition \wedge comp.rightReferenceName = part\}$ **in**

$\underbrace{\texttt{fun }}$ $part\underbrace{\texttt{CompFieldsCD}}cd.name\underbrace{\texttt{:Obj->Obj \{}}$

$\{\!|\underbrace{\texttt{rel[}}c.leftReferenceName\underbrace{\texttt{SubsCD}}cd.name\underbrace{\texttt{,}}$ $c.rightRole\underbrace{\texttt{]}}|\!\}^{\underline{+}}_{c \in comps}$

$\underbrace{\}}$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
fun InsuranceCompFieldsCDcd1:Obj->Obj {
  rel[EmployeeSubsCDcd1, ins]
}
```

Figure 3.9: Rule F4 creates a function for every part of a composite whole-part-relation that returns all linked whole/part instance pairs.

one composition. The function produced for the class `Insurance` returns all tuples of `Employee/Insurance` instances where the first component of each tuple is linked to the second component of the tuple via a link corresponding to the composition association.

3.4 Rules P1 to P4: Classes and Attributes

The rules P1 to P4 are executed for every class diagram *cd* to generate the first part of the body of the Alloy predicate *cd.name* (cf. Figure 3.1) capturing the semantics of the class diagram *cd*.

The translation rules P1 to P4 all use the auxiliary translation rule H1 shown in Figure 3.10. It translates `Type` and `Name` elements of the abstract syntax to corresponding names of functions and signatures defined in the Alloy module. Translation rule H1 is used in translation rules P1 (cf. Figure 3.11) for attribute types and in rules A1 to A4 (cf. Figure 3.15, Figure 3.16, Figure 3.16, Figure 3.17) for associations. In case the name or type given as input to rule H1 refers to an enumeration type, the auxiliary translation rule produces the name of the Alloy function that returns all atoms representing the enumeration type's possible values as defined in the CD *cd*. Similarly, if the input corresponds to a class, the rule generates the name of the Alloy function that returns all atoms representing all instances of subclasses of the class in the CD *cd*. Otherwise, if the input corresponds to an interface, the translation produces the name of the function that returns all atoms representing all instances of classes in the CD *cd* implementing the interface. In any other case, the translation assumes the type or name belongs to a primitive or unknown type and returns the name of the Alloy signature that represents it.

Translation Rule

Translation rule with parameters $cd \in CD$ and $type \in Type \cup Name$:

H1 **if (**$\exists e \in cd.enum : e.name = type$**) then**
$type$EnumCD$cd.name$
else if ($\exists c \in cd.class : c.name = type$**) then**
$type$SubsCD$cd.name$
else if ($\exists i \in cd.interface : i.name = type$**) then**
$type$SubsCD$cd.name$
else
type_$type$

Figure 3.10: Translation rule to translate type names from a CD to corresponding Alloy functions or signatures.

Translation Rule

Translation rule with parameter $cd \in CD$:

P1 $\forall c \in cd.class : \forall a \in \bigcup_{c' \in c.superclasses^*} c'.attribute$:
ObjAttrib[$c.name, a.name,$ **executeRule(**H1 $a.type$ cd**)**]

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjAttrib[Vehicle, licensePlate, type_String]
ObjAttrib[Vehicle, regDate, type_Date]
ObjAttrib[Car, licensePlate, type_String]
ObjAttrib[Car, regDate, type_Date]
ObjAttrib[Insurance, kind, InsuranceKindEnumCDcd1]
ObjAttrib[Driver, exp, DrivingExpEnumCDcd1]
ObjAttrib[Truck, licensePlate, type_String]
ObjAttrib[Truck, regDate, type_Date]
```

Figure 3.11: Rule P1 uses predicate `ObjAttrib` to declare the attributes of every class in the class diagram *cd*.

The rule P1 shown in Figure 3.11 instantiates the predicate `ObjAttrib` (cf. Listing 3.2) to declare the types and multiplicities of the attributes of every class in the class diagram *cd*. The list of attributes also includes all inherited attributes of the class in *cd* since there is no inheritance on the Alloy signature level for the classes of the CD. The translation first flattens and later rebuilds the inheritance hierarchy. In particular, as part of flattening, the complete list of attributes and associations of each class is collected from all its super classes. Given a class c and an attribute a of type t, the generated predicate ensures the `get` relation only relates instances of class c with the field name a to exactly one object of type t. The lower part of Figure 3.11 shows the results produced by the translation given the CD shown in Listing 2.2 as input.

Translation Rule

Translation rule with parameter $cd \in CD$:

$$
\begin{array}{ll}
\text{P2} & \forall c \in cd.class: \\
& \quad \textbf{let } fields = \{a.name \mid \exists c' \in c.superclasses^{*} : a \in c'.attribute\} \cup \\
& \qquad \{a.leftRole \mid a \in cd.association \ \wedge a.rightReferenceName \in \\
& \qquad\quad \{c'.name \mid c' \in c.superclasses^{*} \vee \\
& \qquad\qquad\qquad c' \in c.superclasses^{*}.interfaces^{*}\}\} \cup \\
& \qquad \{a.rightRole \mid a \in cd.association \ \wedge a.leftReferenceName \in \\
& \qquad\quad \{c'.name \mid c' \in c.superclasses^{*} \vee \\
& \qquad\qquad\qquad c' \in c.superclasses^{*}.interfaces^{*}\}\} \ \textbf{in} \\
& \quad \underline{\texttt{ObjFNames[}}c.name\underline{\texttt{,}} \ \textbf{if(}|fields| > 0\textbf{)} \ \textbf{then} \ \{\!|fn|\!\}^{\underline{+}}_{fn \in fields} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else} \ \underline{\texttt{none}} \\
& \quad \underline{\texttt{]}}
\end{array}
$$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjFNames[Vehicle, licensePlate + regDate]
ObjFNames[Company, cars + emps]
ObjFNames[Employee, ins]
ObjFNames[Car, licensePlate + regDate + drivenBy]
ObjFNames[Insurance, kind]
ObjFNames[License, owner]
ObjFNames[Driver, exp + license + drives + ins]
ObjFNames[Truck, licensePlate + regDate]
```

Figure 3.12: Rule P2 restricts the tuples of the `get` relation to the attributes of the class and to the role names of its partners in associations.

The rule P2 defined in Figure 3.12 restricts the `get` relation (cf. Listing 3.1) to only relate `Obj` atoms representing objects with the `FName` atoms corresponding to the field names of the names of the attributes defined in the object's type and to the role names of the types associated with the type of the object. The rule incorporates class and interface inheritance. For each class $c \in cd.class$, the set $fields$ defined in the let/in construct of the rule is defined as the union of three sets. The first set contains the names of all attributes of the class and its superclasses. The second set contains the left role names of all associations where the class, one of its superclasses or one of the interfaces it implements is referenced on the right association end. The third set contains the right role names of all associations where the class, one of its superclasses or one of the interfaces it implements is referenced on the left association end. The translation of the CD given in Listing 2.2 produces the results shown in the lower part of Figure 3.12.

The rule P3 given in Figure 3.13 specifies that all signatures corresponding to abstract classes must have no instances and that signatures representing singleton classes must contain exactly one atom. The CD given in Listing 2.2 does not contain any singleton class but one abstract class called `Vehicle`. The lower part of Figure 3.13 shows the results from applying translation rule P3 to the CD.

Translation Rule

Translation rule with parameter $cd \in CD$:

P3 $\forall c \in \{c \in cd.class \mid c.isAbstract\}:$
$\underbrace{\texttt{no }}c.name$
$\forall c \in \{c \in cd.class \mid \texttt{"singleton"} \in c.stereotype\}:$
$\underbrace{\texttt{one }}c.name$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
no Vehicle
```

Figure 3.13: Rule P3 ensures signatures representing abstract classes have no atoms and that signatures representing singleton classes contain exactly one atom.

Translation Rule

Translation rule with parameter $cd \in CD$:

P4 $\underbrace{\texttt{Obj = }}\ \textbf{if}\ |cd.class| > 0\ \textbf{then}\ \{\!|c.name|\!\}^{\underbrace{+}}_{c \in cd.class}\ \textbf{else}\ \underbrace{\texttt{none}}$

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
Obj = Vehicle + Company + Employee + Car + Insurance +
      License + Driver + Truck
```

Figure 3.14: Rule P4 restricts all objects in object models of the CD to be instances of the classes of the CD.

The rule P4 defined in Figure 3.14 restricts all atoms in Alloy instances representing objects in object models of the CD to be members of the signatures representing the classes of the CD. This constraint is important for multiple CD analysis where the Alloy module might contain signatures extending `Obj` that represent classes of other class diagrams. The lower part of Figure 3.14 shows the results produced by translating the class diagram given in Listing 2.2.

3.5 Rules A1 to A6: Associations

The translation rules A1 to A6 handle the translation of associations. This includes bi- and unidirectional as well as undirected associations, compositions, and the various types of multiplicities on association ends.

The translation rule A1 shown in Figure 3.15 specifies a constraint on the sets of links of bidirectional associations using the predicate `BidiAssoc` (cf. Listing 3.2). The translation result requires all objects that are linked to another object via a bidirectional association also have a link to the other object via the same association. The class diagram given in Listing 2.2, for instance, contains one bidirectional association between the classes `Driver`

Translation Rule

Translation rule with parameter $cd \in CD$:

A1 $\forall a \in \{a \in cd.association \mid a.isBidirectional\}$:
BidiAssoc[**executeRule(**H1 $a.leftReferenceName$ cd**)**,
$a.rightRole$,
executeRule(H1 $a.rightReferenceName$ cd**)**,
$a.leftRole$]

Result of application to the CD cd1 shown in Listing 2.2:

Alloy

```
BidiAssoc[DriverSubsCDcd1,drives,CarSubsCDcd1,drivenBy]
```

Figure 3.15: Rule A1 constraints the sets of links of bidirectional associations.

Translation Rule

Translation rule with parameter $cd \in CD$:

A2 $\forall part \in \{a.RightReferenceName \mid a \in cd.association \wedge a.isComposition\}$:
Composition[$part$CompFieldsCD$cd.name$,
$part$SubsCD$cd.name$]

Result of application to the CD cd1 shown in Listing 2.2:

Alloy

```
Composition[InsuranceCompFieldsCDcd1, InsuranceSubsCDcd1]
```

Figure 3.16: Rule A2 ensures parts of compositions have at most one whole.

and Car. The lower part of Figure 3.15 shows the results of applying the translation rule to the class diagram. For each object having the type or supertype Driver that is connected via a drives link to an object of type or supertype Car, there must be a drivenBy link that connects the same Car object to the same Driver object and vice versa.

The translation rule A2 defined in Figure 3.16 instantiates the predicate Composition (Listing 3.2) for each part of a composition in the CD. The produced constraint ensures the get relation relates at most one atom representing a whole instance to each atom representing a part instance.

The translation rules A3 to A6 shown in Figure 3.17, Figure 3.18, Figure 3.19, and Figure 3.20 handle multiplicities of association ends. Rules A3 and A4 handle the multiplicities of associations that allow navigation from right to left, whereas rules A5 and A6 handle associations that allow navigation from left to right. The translation rules A3 and A5 also instantiate predicates stating multiplicity constraints of bidirectional and undirected associations. Boundaries for the number of existing links are defined using the predicates ObjLUAttrib and ObjLU for lower and upper bounds. The predicates ObjLAttrib and ObjL are used for the specification of lower bounds only. Each of the four predicates is

Translation Rule

Translation rule with parameter $cd \in CD$:

A3 $\forall a \in \{a \in cd.association \mid a.isBidirectional \vee a.isRightToLeft\}$:
  if $a.leftCardinality.isLowerUpper$ **then**
    `ObjLUAttrib[`
      `executeRule(`H1 $a.rightReferenceName$ cd**`)`** `,`
      $a.leftRole$ `,`
      `executeRule(`H1 $a.leftReferenceName$ cd**`)`** `,`
      $a.leftCardinality.lower$ `,` $a.leftCardiality.upper$ `]`
  else
    `ObjLAttrib[`
      `executeRule(`H1 $a.rightReferenceName$ cd**`)`** `,`
      $a.leftRole$ `,`
      `executeRule(`H1 $a.leftReferenceName$ cd**`)`** `,`
      $a.leftCardinality.lower$ `]`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjLUAttrib[CarSubsCDcd1, drivenBy, DriverSubsCDcd1, 1, 1]
ObjLAttrib[CompanySubsCDcd1, emps, EmployeeSubsCDcd1, 0]
ObjLUAttrib[LicenseSubsCDcd1, owner, DriverSubsCDcd1, 1, 1]
```

Figure 3.17: Rule A3 ensures the cardinality constraints stated on the left sides of bidirectional associations, undirected association, and associations that are navigable from right to left are respected.

defined in Listing 3.3. The multiplicity of a cardinality given by a constant k is internally expressed as lower and an upper bound k`..`k. The range k`..`$\star$ is expressed as a lower bound only, and multiplicity $\star$ is expressed as lower bound 0.

Translation rule A3 is defined in Figure 3.17 and applies to bidirectional and undirected associations (concrete syntax `<->` and `--`) as well as to associations that are navigable from right to left (concrete syntax `<-`). The rule ensures the cardinality constraints stated on the left sides of the associations are respected. In case the association defines a lower and an upper bound on its left hand side, the translation rule instantiates the predicate `ObjLUAttrib`. Otherwise, the association does only define a lower bound on its left side. In this case the rule generates an instantiation of the predicate `ObjLAttrib`. The lower part of Figure 3.17 shows the results from transforming the CD depicted in Listing 2.2. The first predicate resulting from the translation originates from the bidirectional association between the classes `Driver` and `Car`. The second predicate is produced for the association between the classes `Employee` and `Company`, which is only navigable from right to left. The last predicate becomes instantiated for the undirected association between the classes `Driver` and `License`.

The rule A4 described in Figure 3.18 only applies to associations that are navigable from right to left (concrete syntax `<-`). It ensures the cardinality constraints stated on the

Translation Rule

Translation rule with parameter $cd \in CD$:

A4 $\forall asc \in \{a \in cd.association \mid a.isRightToLeft\}$:
  if $asc.rightCardinality.isLowerUpper$ **then**
    `ObjLU[`
      `executeRule(`H1 $asc.leftReferenceName\ cd$**`)`** `,`
      $asc.leftRole$ `,`
      `executeRule(`H1 $asc.rightReferenceName\ cd$**`)`** `,`
      $asc.rightCardinality.lower$ `,` $asc.rightCardiality.upper$ `]`
  else
    `ObjL[`
      `executeRule(`H1 $asc.leftReferenceName\ cd$**`)`** `,`
      $asc.leftRole$ `,`
      `executeRule(`H1 $asc.rightReferenceName\ cd$**`)`** `,`
      $asc.rightCardinality.lower$ `]`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjL[EmployeeSubsCDcd1, emps, CompanySubsCDcd1, 0]
```

Figure 3.18: Rule A4 ensures the cardinality constraints stated on the right sides of associations, which are navigable from right to left, are respected.

right sides of the associations are respected. In case the right side of an association is restricted by a lower and an upper bound, the translation rule creates an instantiation of the `ObjLU` predicate, otherwise it instantiates the `ObjL` predicate. The class diagram shown in Listing 2.2 contains exactly one association that is only navigable from right to left. The cardinality on the left side of the association is given by $\star$. Since this cardinality specification only defines a lower bound, the translation produces an instantiation of the `ObjL` predicate as indicated in the lower part of Figure 3.18.

Rule A5 given in Figure 3.19 is defined analogously to rule A3 and applies to bidirectional and undirected associations (concrete syntax <-> and --) as well as to associations defined from left to right (concrete syntax ->). Rule A6 given in Figure 3.20 is the analog to rule A4. In contrast to rule A4, it applies to associations that are navigable from left to right instead of to associations that allow navigation from right to left. The lower parts of Figure 3.19 and Figure 3.20 show the results from applying the translations to the CD given in Listing 2.2.

Translation Rule

Translation rule with parameter $cd \in CD$:

A5 $\forall asc \in \{a \in cd.association \mid a.isBidirectional \vee a.isLeftToRight\}$:
if $asc.rightCardinality.isLowerUpper$ **then**
`ObjLUAttrib[`
`executeRule(`H1 $asc.leftReferenceName\ cd$**`)`** `,`
$asc.rightRole$ `,`
`executeRule(`H1 $asc.rightReferenceName\ cd$**`)`** `,`
$asc.rightCardinality.lower$ `,`
$asc.rightCardiality.upper$ `]`
else
`ObjLAttrib[`
`executeRule(`H1 $asc.leftReferenceName\ cd$**`)`** `,`
$asc.rightRole$ `,`
`executeRule(`H1 $asc.rightReferenceName\ cd$**`)`** `,`
$asc.rightCardinality.lower$ `]`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjLUAttrib[DriverSubsCDcd1,drives,CarSubsCDcd1,0,1]
ObjLAttrib[CompanySubsCDcd1,cars,CarSubsCDcd1,0]
ObjLUAttrib[EmployeeSubsCDcd1,ins,InsuranceSubsCDcd1,1,1]
ObjLUAttrib[DriverSubsCDcd1,license,LicenseSubsCDcd1,0,3]
```

Figure 3.19: Rule A5 ensures the cardinality constraints stated on the right sides of bidirectional associations, undirected association, and associations that are navigable from right to left are respected.

Translation Rule

Translation rule with parameter $cd \in CD$:

A6 $\quad \forall asc \in \{a \in cd.association \mid a.isLeftToRight\}$:

if $asc.leftCardinality.isLowerUpper$ **then**

`ObjLU[`

`executeRule(`H1 $asc.rightReferenceName\ cd$**`)`**`,`

$asc.rightRole$`,`

`executeRule(`H1 $asc.leftReferenceName\ cd$**`)`**`,`

$asc.leftCardinality.lower$`,`

$asc.leftCardiality.upper$ `]`

else

`ObjL[`

`executeRule(`H1 $asc.rightReferenceName\ cd$**`)`**`,`

$asc.rightRole$`,`

`executeRule(`H1 $asc.leftReferenceName\ cd$**`)`**`,`

$asc.leftCardinality.lower$ `]`

Result of application to the CD `cd1` shown in Listing 2.2:

Alloy

```
ObjLU[CarSubsCDcd1, cars, CompanySubsCDcd1, 0, 1]
ObjLU[InsuranceSubsCDcd1, ins, EmployeeSubsCDcd1, 1, 1]
```

Figure 3.20: Rule A6 ensures the cardinality constraints stated on the left sides of associations that are navigable from left to right are respected.

Chapter 4

Translation and Analysis of Multiple CDs

The previous chapter introduced the translation rules for translating a set CD of UML/P class diagrams [Rum16] into a single Alloy module. For each CD $cd \in CD$ the output of the translation contains the predicate *cd.name* that expresses the semantics of the CD in terms of Alloy instances representing object models. Running one of these predicates using Alloy's build in `run` command presents Alloy instances to the user that represent object models that are members of the semantics of the corresponding CD. Besides calculating the object models in the semantics of a single CD, different predicates can also be combined to answer more complex analysis questions such as whether there are object models in the semantics of one CD that are no members of the semantics of another CD. Based on this question [MRR11b] presented *cddiff*, a semantic differencing operator for CDs based on the translation presented in this technical report. Given two class diagrams $cd_1, cd_2 \in CD$, $\mathit{cddiff}(cd_1, cd_2)$ is defined as the set of object models possible in cd_1 that are not possible in cd_2. [MRR11b] specifically defined a bounded version of the operator $\mathit{cddiff}_k(cd_1, cd_2)$, which only includes object models where the number of instances of each class is not greater than an arbitrary but fixed scope k. Technically, to compute $\mathit{cddiff}_k(cd_1, cd_2)$, [MRR11b] suggests using a translation to Alloy similar to the translation presented in this report. The translation described in this report is a slightly enhanced version of the translation employed by [MRR11b]. Given a set $CD = \{cd_1, cd_2\}$ of two class diagrams, computing Alloy instances representing the set of object models in $\mathit{cddiff}_k(cd_1, cd_2)$ can be done by running the predicate `diff` defined as:

$$\begin{array}{l}\texttt{pred diff \{} \\ \quad cd_1.\mathit{name}\ \texttt{and not}\ cd_2.\mathit{name} \\ \texttt{\}}\end{array}$$

Another interesting analysis questions is whether a class diagram $cd_1 \in CD$ refines another class diagram $cd_2 \in CD$, i.e., whether all object models in the semantics of cd_1 are also members of the semantics of cd_2. Reformulating the problem statement leads to the question whether there are object models in the semantics of cd_1 that are not contained in the semantics of cd_2. If such an object model does not exist, then cd_1 refines cd_2. For an arbitrary but fixes scope k this question can be easily answered by running the predicate `diff` as defined above.

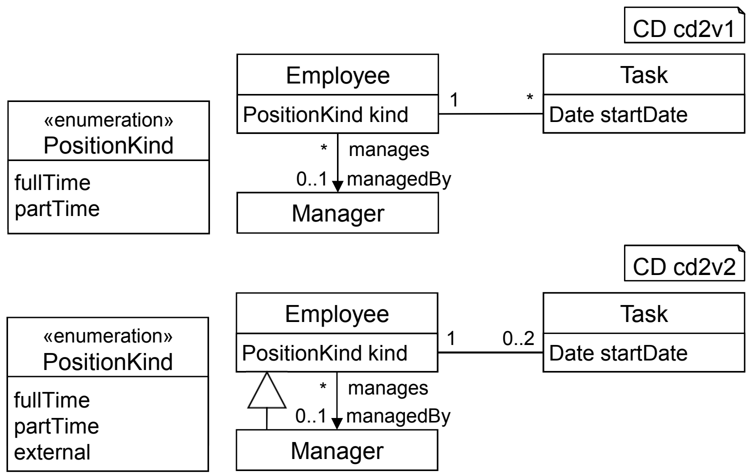

Figure 4.1: Two syntactically similar CDs *cd2v1* and *cd2v2*.

alloy

```
// Rule U1
sig Task extends Obj {}
sig Employee extends Obj {}
sig Manager extends Obj {}
// Rule U2
private one sig task extends FName {}
private one sig managedBy extends FName {}
private one sig kind extends FName {}
private one sig manages extends FName {}
private one sig employee extends FName {}
private one sig startDate extends FName {}
// Rule U3
private one sig type_Date extends Val {}
// Rule U4
private one sig enum_PositionKind_partTime extends EnumVal {}
private one sig enum_PositionKind_external extends EnumVal {}
private one sig enum_PositionKind_fullTime extends EnumVal {}
```

Listing 4.1: Output of the translation rules U1-U4 given the CDs *cd2v1* *cd2v2* depicted in Figure 4.1 as input.

This chapter demonstrates the analysis of multiple CDs by example. Figure 4.1 shows the graphical representation of the two syntactically rather similar class diagrams *cd2v1* and *cd2v2*, which were previously presented in [MRR11b]. The CD *cd2v1* contains the enumeration type `PositionKind`, the three classes `Employee`, `Manager`, and `Task` as well as two associations. The CD *cd2v2* is a revised version of the CD *cd2v1*. An additional inheritance relation between the classes `Manager` and `Employee` has been added, one multiplicity on the association between the classes `Employee` and `Task` has been changed from $\star$ to `0..2`, and the value `external` has been added to the enumeration type `PositionKind`.

Listing 4.1, Listing 4.2, Listing 4.3, and Listing 4.4 show the non-generic part of the Alloy module produced by the translation presented in Chapter 3 when given the two CDs *cd2v1* and *cd2v2* shown in Figure 4.1 as input. Translation rules U1 to U4 produce signatures for

alloy

```
1 // Rule F1
2 fun TaskSubsCDcd2v1: set Obj  { Task }
3 fun EmployeeSubsCDcd2v1: set Obj { Employee }
4 fun ManagerSubsCDcd2v1: set Obj { Manager }
5 fun TaskSubsCDcd2v2: set Obj { Task }
6 fun EmployeeSubsCDcd2v2: set Obj { Employee + Manager }
7 fun ManagerSubsCDcd2v2: set Obj { Manager }
8 // Rule F3
9 fun PositionKindEnumCDcd2v1: set EnumVal  {
10   enum_PositionKind_partTime +
11   enum_PositionKind_fullTime
12 }
13 fun PositionKindEnumCDcd2v2: set EnumVal  {
14   enum_PositionKind_partTime +
15   enum_PositionKind_external +
16   enum_PositionKind_fullTime
17 }
```

Listing 4.2: Output of the translation rules F1-F4 given the CDs *cd2v1* *cd2v2* depicted in Figure 4.1 as input.

CD elements collected from both CDs, which are referenced by the generated parts specific to each of the CDs. The results from applying rules U1 to U4 are given in Listing 4.1. Listing 4.2 depicts the parts generated by translation rules F1-F4. The functions given in lines 2-4 and in lines 9-12 are generated from *cd2v1*, whereas the functions shown in lines 5-7 and lines 13-17 are produced for *cd2v2*.

The predicate `cd2v1` encoding the semantics of *cd2v1* is depicted in Listing 4.3 and the predicate `cd2v2` encoding the semantics of the CD *cd2v2* is given in Listing 4.4. The syntactic differences of the CDs induce several differences in the generated predicates. The enumeration type `PositionKind` in *cd2v2* consists of one more enumeration value than the same named enumeration type in *cd2v1*. Thus, the value range of fields of type `PositionKind` must differ in both CDs, which is reflected in the predicate *cd2v1* in line 4 and in predicate `cd2v2` in line 4. The functions `PositionKindEnumCDcd2v1` and `PositionKindEnumCDcd2v2` return different sets (cf. Listing 4.2, ll. 9-17). In class diagram *cd2v1* the class `Manager` has no attributes, which is reflected in the predicate *cd2v1* in line 7. In contrast, the additional inheritance relation in *cd2v2* leads to class `Manager` inheriting all attributes from class `Employee`, which has two effects on the difference between the two predicates. First, in *cd2v2* class `Manager` inherits the attribute `kind` having the enumeration type `PositionKind` from class `Employee`. Thus, predicate `cd2v2` additionally specifies the value range for this attribute (cf. Listing 4.4, l. 5). Second, the inherited field names are added to the field names that can be related to the signature `Manager` via the `get` relation (cf. Listing 4.4, l. 8). The predicate instantiations generated for the two associations in the CDs differ since the subclass relations of all classes in both CDs are treated individually (cf. Listing 4.3, ll. 10-17 and Listing 4.4, ll. 11-18). However, class `Employee` is the only class a subclass has been added for in *cd2v2*. Thus, as can be seen when comparing the generated functions for expressing subclassing (cf. Listing 4.2, ll. 2-7), only the corresponding subclassing functions generated for the `Employee` classes differ in functionality (cf. Listing 4.2, l. 3 and l. 6). Further, the cardinality `*` on one side of the undirected association between the classes `Employee` and `Task` changed to `0..2`. While the former cardinality only induces a lower bound of the

alloy

```
1 pred cd2v1 {
2   // Rules P1 - P4
3   ObjAttrib[Task, startDate, type_Date]
4   ObjAttrib[Employee, kind, PositionKindEnumCDcd2v1]
5   ObjFNames[Task,  startDate +  employee +  none]
6   ObjFNames[Employee,  kind +  task +  managedBy +  none]
7   ObjFNames[Manager,  none]
8   Obj = (Task + Employee + Manager)
9   // Rules A1 - A6
10  BidiAssoc[EmployeeSubsCDcd2v1, task, TaskSubsCDcd2v1, employee]
11  ObjLUAttrib[EmployeeSubsCDcd2v1, managedBy, ManagerSubsCDcd2v1, 0, 1]
12  ObjLUAttrib[TaskSubsCDcd2v1, employee, EmployeeSubsCDcd2v1, 1, 1]
13  ObjLAttrib[EmployeeSubsCDcd2v1, task, TaskSubsCDcd2v1, 0]
14  ObjL[ManagerSubsCDcd2v1, managedBy, EmployeeSubsCDcd2v1, 0]
15 }
```

Listing 4.3: The predicate `cd2v1` produced by the CD2Alloy translation. Translation rules P1-P4 and A1-A6 produce the body of the predicate when given the CD *cd2v1* (cf. Figure 4.1) as input.

alloy

```
pred cd2v2 {
  // Rules P1 - P4
  ObjAttrib[Task, startDate, type_Date]
  ObjAttrib[Employee, kind, PositionKindEnumCDcd2v2]
  ObjAttrib[Manager, kind, PositionKindEnumCDcd2v2]
  ObjFNames[Task,  startDate +  employee +  none]
  ObjFNames[Employee,  kind +  task +  managedBy +  none]
  ObjFNames[Manager,  kind +  task +  managedBy +  none]
  Obj = (Task + Employee + Manager)
  // Rules A1 - A6
  BidiAssoc[EmployeeSubsCDcd2v2, task, TaskSubsCDcd2v2, employee]
  ObjLUAttrib[EmployeeSubsCDcd2v2, managedBy, ManagerSubsCDcd2v2, 0, 1]
  ObjLUAttrib[TaskSubsCDcd2v2, employee, EmployeeSubsCDcd2v2, 1, 1]
  ObjLUAttrib[EmployeeSubsCDcd2v2,task, TaskSubsCDcd2v2, 0, 2]
  ObjL[ManagerSubsCDcd2v2, managedBy, EmployeeSubsCDcd2v2, 0]
}
```

Listing 4.4: The predicate `cd2v2` produced by the CD2Alloy translation. Translation rules P1-P4 and A1-A6 produce the body of the predicate when given the CD *cd2v2* (cf. Figure 4.1) as input.

`Task` instances associated with an `Employee` instance, the latter induces a lower and an upper bound (cf. Listing 4.3, l. 16 and Listing 4.3, l. 17). Different analyses of the Alloy module generated from the translation can be performed in the Alloy Analyzer using run commands. For instance, executing the command `run {cd2v1 and not cd2v2} for 10` instructs Alloy to compute instances with at most ten atoms per signature that satisfy the generated predicate `cd2v1` and not the generated predicate `cd2v2`.

Chapter 5

Translation of Alloy Instances to Object Diagrams

The intermediate results of CD analyses using our translation are Alloy instances of the Alloy modules generated from the input CDs. Alloy instances in scope k exist if, and only if, object models with up to k objects exist (scope k bounds the number of Alloy atoms in relation `Obj` representing objects in object models). The computed Alloy instances represent object models that are possible in the input CDs. This chapter presents a translation from Alloy instances to UML/P object diagrams (ODs) [Rum16]. Each OD resulting from such a translation represents the object model corresponding to an instance computed by the Alloy analyzer. Such object diagrams can be presented to engineers as witnesses of analysis results. The resulting UML/P ODs consist of objects with attributes and links between the objects. The MontiCore grammar for the object diagrams used in this report is given in [Sch12].

Section 5.1 describes the structure of Alloy instances of modules resulting from the CD2Alloy translation. The structure is independent of specific CDs. Based on the common structure, Section 5.2 presents the translation from Alloy instances to UML/P ODs.

5.1 Structure of CD2Alloy Alloy instances

An Alloy instance consists of the signatures, fields, and relations that are specified in its corresponding Alloy module. Each signature is a set of atoms that contains at most as many elements as defined by it's user specified scope. Each atom can be interpreted as a signature instance. Fields and relations are translated into tuples of atoms. Since Alloy instances are always finite, the sets and tuples of atoms are always finite, too. Independent of the input CDs given to the CD2Alloy translation, every generated Alloy module includes the signatures, fields, and relations specified in the generic part of the translation described in Section 3.2.1. Thus all generated Alloy modules share the signature `Obj` with it's field `get` and the signatures `FName`, `Val`, and `EnumVal` (cf Listing 3.1). The shared signatures induce a common form for all Alloy instances computed for any module produced by the translation. The elements of the common form are used for the translation to ODs. Every Alloy instance computed for any Alloy module produced by the translation for any set of input CDs consists of the following sets of atoms:

Alloy

```
FName = {employee$0, kind$0, managedBy$0,
         manages$0, startDate$0, task$0}
EnumVal = {enum_PositionKind_external$0,
           enum_PositionKind_fullTime$0,
           enum_PositionKind_partTime$0}
Val = {type_Date$0}
Obj = {Employee$0, Manager$0, Task$0, Task$1, Task$2}
get = {Employee$0->kind$0->enum_PositionKind_fullTime$0,
       Employee$0->managedBy$0->Manager$0,
       Employee$0->task$0->Task$0,
       Employee$0->task$0->Task$1,
       Employee$0->task$0->Task$2,
       Task$0->employee$0->Employee$0,
       Task$0->startDate$0->type_Date$0,
       Task$1->employee$0->Employee$0,
       Task$1->startDate$0->type_Date$0,
       Task$2->employee$0->Employee$0,
       Task$2->startDate$0->type_Date$0}
```

Listing 5.1: Excerpt of an Alloy instance for the module generated from the class diagrams $cd2v1$ and $cd2v2$ depicted in Figure 4.1.

- Obj representing all objects in the OM,
- Val representing values of primitive and unknown types of attributes in the OM,
- $EnumVal$ representing enumeration values assigned to attributes in the OM,
- $FName$ representing attribute and role names,

and the relation

- $get \subseteq (Obj \times FName \times (Obj \cup Val \cup EnumVal))$ representing links and attribute assignments in the OM.

Figure 4.1 shows an excerpt of an Alloy instance of the module produced from the CD2Alloy translation for CDs $cd2v1$ and $cd2v2$ (cf. Listing 5.1) in the textual Alloy syntax for instances. It shows all the common Alloy signatures and relations as described above.

5.2 Translation rules

The translation of Alloy instances to UML/P ODs is done with translation rule O1 shown in Figure 5.5. The name of each atom in the sets Obj, Val, $FName$, and $EnumVal$ is composed of the name of the atom's most specific signature, followed by the symbol `$`, and an integer uniquely identifying the atom under the atoms in the set of its signature (cf. Listing 5.1). Translation rule O1 utilizes the auxiliary translation rules defined in Figure 5.1, Figure 5.2, Figure 5.3, and Figure 5.4 for producing declarations of objects, attributes of primitive types, attributes of enumeration types, and links. The auxiliary translation rules internally compute type and instance names for objects in the resulting OD. All auxiliary translations are defined by means of pattern matching on the string

Translation Rule

Translation rule with parameter $o \in Obj$:

ODecl $\;\;$ **let** $name$ = THE $x \in \{sig\frown num \mid o = sig\frown \$ \frown num\}$ **in**
let $type$ = THE $x \in \{sig \mid \exists num : o = sig\frown \$ \frown num\}$ **in**
$name$: $type$

Result of application to the `Obj` atom name `Employee$0`:

OD
```
1 Employee0:Employee
```

Figure 5.1: Rule `ODecl` translates an `Obj` atom to an object declaration.

Translation Rule

Translation rule with parameters $v \in Val$ and $a \in FName$:

Prim $\;\;$ **let** $type$ = THE $x \in \{t \mid \exists num : v = \texttt{type\_}\frown t\frown \$ \frown num\}$ **in**
let $name$ = THE $x \in \{sig \mid \exists num : a = sig\frown \$ \frown num\}$ **in**
let val = THE $x \in \{sig \mid \exists num : v = sig\frown \$ \frown num\}$ **in**
$type$ $name$ = some_val

Result of application to the `Val` atom name `type_Date$0` and the `FName` atom name `startDate$0`:

OD
```
1 Date startDate = some_type_Date
```

Figure 5.2: Translation rule `Prim` translates a `Val` atom and a `FName` atom to an attribute declaration. The name attribute's name is encoded by the `FName` atom, whereas the value and the type of the attribute are encoded by the `Val` atom.

representation of atom names where the symbol $\frown$ denotes string concatenation. The auxiliary translation rules use the `THE` operator. In this context, the operator retrieves the unique element of a singleton set, i.e., given a singleton set $S = \{e\}$ the expression `THE` $x \in S$ retrieves the unique element e of S and binds its value to x. For simplicity we assume the underscore (_) and dollar ($) symbols must not be used in names of elements in the CDs given as input for the CD2Alloy translation. This assumption leads to the uniqueness of the single elements contained in the sets described in the auxiliary translation rules.

The rule `ODecl` depicted in Figure 5.1 transforms *Obj* atom names to expressions introducing object declarations. It computes an object's name by dropping the $ symbol from the atom's name. The type of an object is given by the name of the atom's signature. The rule `ODecl` translates the atom name `Employee$0`, for instance, to `Employee0:Employee`.

The translation rule `Prim` shown in Figure 5.2 takes as input a `Val` and a `FName` atom. It produces an attribute declaration for an attribute of primitive or unknown type. The value and the type of the attribute are encoded in the name of the `Val` atom. The attribute's name is encoded by the `FName` atom's name. The lower part of Figure 5.2 shows an example for the application of the translation rule.

`Translation Rule`

Translation rule with parameters $e \in EnumVal$ and $a \in FName$:

$$
\begin{array}{ll}
\text{Enum} & \textbf{let } type = \texttt{THE } x \in \{t \mid \exists v : \exists n : e = \texttt{enum\_}{}^\frown t^\frown\texttt{\_}{}^\frown v^\frown\texttt{\$}^\frown n\} \textbf{ in} \\
 & \textbf{let } name = \texttt{THE } x \in \{sig \mid \exists num : a = sig^\frown\texttt{\$}^\frown num\} \textbf{ in} \\
 & \textbf{let } val = \texttt{THE } x \in \{v \mid \exists t : \exists n : e = \texttt{enum\_}{}^\frown t^\frown\texttt{\_}{}^\frown v^\frown\texttt{\$}^\frown n\} \textbf{ in} \\
 & type\texttt{␣}name\texttt{␣=␣}val
\end{array}
$$

Result of application to the `FName` atom name `kind$0` and the `EnumVal` atom name `enum_PositionKind_external$0`:

`OD`

```
1 PositionKind kind = external
```

Figure 5.3: Translation rule `Enum` translates an `EnumVal` atom and a `FName` atom to a declaration of an attribute having the name of the field encoded by the `FName` atom and having the type encoded by the `EnumVal` atom.

`Translation Rule`

Translation rule with parameters $o \in Obj$, $o' \in Obj$, and $a \in FName$:

$$
\begin{array}{ll}
\text{Link} & \textbf{let } lObj = \texttt{THE } x \in \{sig^\frown num \mid o = sig^\frown\texttt{\$}^\frown num\} \textbf{ in} \\
 & \textbf{let } lnkName = \texttt{THE } x \in \{sig \mid \exists num : a = sig^\frown\texttt{\$}^\frown num\} \textbf{ in} \\
 & \textbf{let } rObj = \texttt{THE } x \in \{sig^\frown num \mid o' = sig^\frown\texttt{\$}^\frown num\} \textbf{ in} \\
 & \texttt{link␣}lObj\texttt{␣->␣(}lnkName\texttt{)␣}rObj
\end{array}
$$

Result of application to the `Obj` atom names `Employee$0` and `Task$1` and the `FName` atom name `task$0`:

`OD`

```
1 link Employee0 -> (task) Task1
```

Figure 5.4: Translation rule `Link` translated two `Obj` atoms and one `FName` atom to a link declaration. The link connects the objects encoded by the `Obj` atoms and has a name that is encoded by the `FName` atom.

The translation rule `Enum` given in Figure 5.3 produces declarations for attributes of enumeration types from `EnumVal` atom names and `FName` atom names. Enum types and values are derived from `EnumVal` atoms, whereas attribute names are derived from `FName` atoms. For example, the rule `Enum` translates the `EnumVal` atom name `enum_PositionKind_external$0` and the `FName` atom name `kind$0` to `PositionKind kind = external`.

The translation rule `Link` shown in Figure 5.4 creates a link declaration from two `Obj` atom names and a `FName` atom name. The names of the objects connected by the link are encoded in the `Obj` atom names, whereas the `FName` atom name encodes the link name. The rule translates the `Obj` atom names `Employee$0`, `Task$1` and the `FName` atom name `task$0`, for instance, to the expression `link Employee0 -> (task) Task1`.

Translation rule $O1$ defined in Figure 5.5 is parametrized with the sets of atoms Obj, Val, $EnumVal$, and $FName$ of an Alloy instance computed from a module produced by the CD2Alloy translation. It first declares an OD having the name `od`. Inside the body of the OD definition, the rule uses two outer quantifications. The first outer quantification ranges over the set of atoms in the set Obj representing objects in the object model. For each object contained in the set Obj, the rule defines an object. The name of the type of the object is equal to the name of the signature of the atom. The name of the object is equal to the name of the signature of the atom postfixed with the atom's identifier. The outer quantification contains two nested quantifications. The first nested quantification ranges over all tuples in the `get` relation where the first component is equal to the atom bounded by the outer quantification and the last component is a member of the set `Val`. As a result the iteration declares an attribute for each attribute of primitive or undefined type of the object represented by the atom bounded by the outer quantification. Similarly, the second nested quantification declares an attribute for each attribute of an enumeration type. The second outer quantification instantiates all links between objects. It iterates over all pairs of atoms contained in the relation *get* that relate three atoms representing two object and an association name. The result from translating the Alloy instance given in Listing 5.1 is shown in the lower part of Figure 5.5.

Translation Rule

Translation rule with sets of atoms Obj, Val, $EnumVal$, $FName$ and a set $get \subseteq (Obj \times FName \times (Obj \cup Val \cup EnumVal))$ as parameters:

O1 `objectdiagram od {`

$\forall o \in Obj :$

executeRule(ODecl o**)** `{`

$\forall (o', a, val) \in \{(o', a, val) \in get \mid o' = o \wedge val \in Val\} :$

executeRule(Prim val a**)** `;`

$\forall (o', a, val) \in \{(o', a, val) \in get \mid o' = o \wedge val \in EnumVal\} :$

executeRule(Enum val a**)** `;`

`}`

$\forall (o, a, o') \in \{(o, a, o') \in get \mid o \in Obj \wedge o' \in Obj\} :$

executeRule(Link o o' a**)** `;`

`}`

Result of application to the Alloy instance textually shown in Listing 5.1:

OD

```
objectdiagram od {
  Employee0:Employee { PositionKind kind = fullTime; }
  Manager0:Manager {}
  Task0:Task { Date startDate = some_type_Date; }
  Task1:Task { Date startDate = some_type_Date; }
  Task2:Task { Date startDate = some_type_Date; }
  link Employee0 -> (managedBy) Manager0;
  link Employee0 -> (task) Task0;
  link Employee0 -> (task) Task1;
  link Employee0 -> (task) Task2;
  link Task0 -> (employee) Employee0;
  link Task1 -> (employee) Employee0;
  link Task2 -> (employee) Employee0;
}
```

Figure 5.5: Translation of Alloy instances representing object models in the semantics of a CD to UML/P ODs.

Chapter 6

Related Work

UML2Alloy [ABGR07b, ABGR09] is another translation from CDs to Alloy. The transformation is defined by means of a formal CD metamodel, an Alloy metamodel, and transformation rules between metamodel instances. The shallow nature of the translation limits the set of CD features supported by the translation. For instance, multiple inheritance cannot be directly represented as Alloy does not support multiple inheritance. The translation presented in this technical report address this challenge using a deeper embedding strategy that bridges some of the differences between the CD and the Alloy modeling languages.

A formalization of UML package merge is given in [DDZ08]. Unlike our approach, the the method in [DDZ08] does not present a generic transformation to Alloy and does not discuss the analysis of multiple CDs. Another translation of the UML metamodel to Alloy is presented in [Sen10]. Similar to the translation given in this technical report, the translation is not shallow and handles an extended list of CD features such as multiple inheritance and composition. In contrast to our translation, it does not support analyses of multiple input models such as refinement checking.

UMLtoCSP [CCR07] is a translation from UML/OCL to constraint satisfaction problems. Using a constraint solver, the resulting problems can be solved within a user-defined bounded search space. The tool checks for various kinds of satisfiability (and other analysis problems), and can generate example instances (object model) as analysis results. The analyses in CD2Alloy and UMLtoCSP are both fully automated and limited to a bounded scope. By adding further constraints to a constraint satisfaction problem resulting from the UMLtoCSP translation, it should be possible to extend UMLtoCSP to check for Boolean expressions over CDs, as supported by our work.

USE [GBR07] supports the analysis of a CD with OCL invariants. It supports multiple inheritance. Valid instances are searched for enumeratively, within a user-given bounded scope. Other work by the same group, e.g., [SWK+10], report on analyzing UML/OCL models directly using a SAT solver. Analyses of multiple CDs such as checking refinement between two CDs, as available in our work, are not available in any of the works related to USE.

Further examples for the kinds of analyses enabled by the translation given in this report are presented in [MRR11b, MRR11c]. A semantic differencing operator for class diagrams is presented in [MRR11b]. It is implemented using the translation to Alloy presented in this report. A variant of our translation is used in [MRR11c] to support semantic variability in CD/OD consistency analysis. This tool takes three artifacts as input: a CD, an OD, and a feature configuration that specifies choices over a set of semantic variability points. With this, the semantics mapping is configured and analysis results change according to the semantics induced by the selected feature configuration.

Chapter 7

Conclusion

This technical report presented CD2Alloy, a translation from UML CDs to Alloy and back from Alloy instances to UML ODs. It supports many CD language features, including, e.g., directed associations, composite aggregations, interfaces, multiple inheritance, and enumerations. An important feature of the translation is the ability to analyze multiple class diagrams within one Alloy module, which is not possible with previous translations. The ideas and translation rules are demonstrated with running examples.

One future work direction consists of the investigation of encoding additional fragments of UML into Alloy, in order to support additional language features and analyses. Another future work direction consists of integrating OCL into the translation. Additional future work could investigate adding summarization and abstraction strategies to the translation for computing informative, small, and comprehensive object models. Finally, another future work direction is to address the scope limitation and attempt to improve the performance of the analysis in general by using model slicing techniques, tailored to CD analysis. Such optimization may need to take into account also the special cases of analyses involving more than one CDs.

Bibliography

[ABGR07a] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2007.

[ABGR07b] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems*, MODELS'07, pages 436–450. Springer, 2007.

[ABGR09] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 2009.

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling*, 9(1):69–86, 2010.

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[CCR07] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*, pages 547–548. ACM, 2007.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[DDZ08] Jürgen Dingel, Zinovy Diskin, and Alanna Zito. Understanding and improving uml package merge. *Software and Systems Modeling*, 7(4):443–467, October 2008.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[GR99] Martin Gogolla and Mark Richters. Transformation Rules for UML Class Diagrams. In Jean Bézivin and Pierre-Alain Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop,* *Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 92–106. Springer, 1999.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[MGB04] Tiago Massoni, Rohit Gheyi, and Paulo Borba. A UML Class Diagram Analyzer. In *3rd Int. Work. on Critical Systems Development with UML (CSDUML), affiliated with UML Conf.*, pages 143–153, 2004.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class diagrams analysis using Alloy revisited. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *MoDELS*, volume 6981 of *Lecture Notes in Computer Science*, pages 592–607. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic differencing for class diagrams. In Mira Mezini, editor, *ECOOP*, volume 6813 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[OMG15] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.5 (15-03-01), March 2015. `http://www.omg.org/spec/UML/2.5/PDF/` [Online; accessed 2016-11-07].

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[SAB09] Adel Smeda, Adel Alti, and Abbdellah Boukerram. An Environment for Describing Software Systems. *WSEAS Transactions on Computers*, 8(9):1610–1619, 2009.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[Sen10] Sagar Sen. *Automatic Effective Model Discovery*. PhD thesis, Univ. of Rennes, 2010.

[SWK+10] Mathias Soeken, Robert Wille, Mirco Kuhlmann, Martin Gogolla, and Rolf Drechsler. Verifying UML/OCL models using Boolean satisfiability. In *DATE*, pages 1341–1344. IEEE, 2010.

Appendix A

Complete CD2Alloy Translation Example

This appendix contains the complete translation result from translating the class diagram described in Section 2.2. The class diagram is graphically depicted in Figure 2.1. Its textual representation is given in Listing 2.2. The result originated from applying the translation rules defined in Chapter 3.

Alloy

```
module umlp2alloy/cd1_module

// ***** Generic Part *****

//Names of fields/associations in classes of the model
private abstract sig FName {}

//Names of enum values in enums of the model
private abstract sig EnumVal {}

//Values of fields
private abstract sig Val {}

//Parent of all classes relating fields and values
abstract sig Obj {
  get : FName -> { Obj + Val + EnumVal }
}

pred ObjFNames[objs: set Obj, fNames:set  FName] {
  no objs.get[FName - fNames]
}

pred ObjAttrib[objs: set Obj, fName:one FName, fType: set { Obj + Val +
    EnumVal }] {
  objs.get[fName] in fType
  all o: objs | one o.get[fName]
}

pred ObjMeth[objs: set Obj, fName: one FName, fType: set { Obj + Val +
    EnumVal }] {
  objs.get[fName] in fType
  all o: objs | one o.get[fName]
```

```
}

pred ObjLUAttrib[objs: set Obj, fName:one FName, fType: set Obj, low: Int, up: Int] {
  ObjLAttrib[objs, fName, fType, low]
  ObjUAttrib[objs, fName, fType, up]
}

pred ObjLAttrib[objs: set Obj, fName:one FName, fType: set Obj, low: Int] {
  objs.get[fName] in fType
  all o: objs | (#o.get[fName] >= low)
}

pred ObjUAttrib[objs: set Obj, fName:one FName, fType: set Obj, up: Int] {
  objs.get[fName] in fType
  all o: objs | (#o.get[fName] =< up)
}

pred ObjLU[objs: set Obj, fName:one FName, fType: set Obj, low: Int, up: Int] {
  ObjL[objs, fName, fType, low]
  ObjU[objs, fName, fType, up]
}

pred ObjL[objs: set Obj, fName:one FName, fType: set Obj, low: Int] {
  all r: objs | # { l: fType | r in l.get[fName]} >= low
}

pred ObjU[objs: set Obj, fName:one FName, fType: set Obj, up: Int] {
  all r: objs | # { l: fType | r in l.get[fName]} =< up
}

pred BidiAssoc[left: set Obj, lFName:one FName, right: set Obj, rFName:one FName] {
  all l: left | all r: l.get[lFName] | l in r.get[rFName]
  all r: right | all l: r.get[rFName] | r in l.get[lFName]
}

pred Composition[compos: Obj->Obj, right: set Obj] {
  all r: right | lone compos.r
}

fun rel[wholes: set Obj, fn: FName]  : Obj->Obj {
  {o1:Obj,o2:Obj|o1->fn->o2 in wholes <: get}
}

fact NonEmptyInstancesOnly {
  some Obj
}

// ***** Structures common to both CDs *****

// Concrete names of fields in cd
private one sig owner extends FName {}
private one sig cars extends FName {}
private one sig license extends FName {}
private one sig licensePlate extends FName {}
private one sig emps extends FName {}
private one sig drives extends FName {}
private one sig kind extends FName {}
```

```
private one sig of extends FName {}
private one sig regDate extends FName {}
private one sig exp extends FName {}
private one sig drivenBy extends FName {}
private one sig ins extends FName {}

// Concrete value types in model cd
private one sig type_Date extends Val {}
private one sig type_String extends Val {}

// Concrete enum values in model cd
private one sig enum_InsuranceKind_international extends EnumVal {}
private one sig enum_DrivingExp_expert extends EnumVal {}
private one sig enum_InsuranceKind_transport extends EnumVal {}
private one sig enum_DrivingExp_beginner extends EnumVal {}

// Classes and interfaces in model cd
sig Vehicle extends Obj {}
sig Company extends Obj {}
sig Employee extends Obj {}
sig Car extends Obj {}
sig Insurance extends Obj {}
sig License extends Obj {}
sig Driver extends Obj {}
sig Truck extends Obj {}

// ***** CD cd1 *****

// Types wrapping subtypes
fun VehicleSubsCDcd1: set Obj  {
  Vehicle + Car + Truck
}
fun CompanySubsCDcd1: set Obj  {
  Company
}
fun EmployeeSubsCDcd1: set Obj  {
  Employee + Driver
}
fun CarSubsCDcd1: set Obj  {
  Car
}
fun InsuranceSubsCDcd1: set Obj  {
  Insurance
}
fun LicenseSubsCDcd1: set Obj  {
  License
}
fun DriverSubsCDcd1: set Obj  {
  Driver
}
fun TruckSubsCDcd1: set Obj  {
  Truck
}

// Types containing subtypes for definition of associations
fun DriveableSubsCDcd1: set Obj  {
    Vehicle + Car + Truck
}

// Relations that represent compositions which the class is a part of
```

```
fun InsuranceCompFieldsCDcd1: Obj->Obj {
 rel[EmployeeSubsCDcd1, ins]
}

// Enums
// Enum values in cd
fun DrivingExpEnumCDcd1: set EnumVal  {
    enum_DrivingExp_expert + enum_DrivingExp_beginner
}

fun InsuranceKindEnumCDcd1: set EnumVal  {
    enum_InsuranceKind_international + enum_InsuranceKind_transport
}


// Values and relations in cd
pred cd1 {

  // Definition of class Vehicle
  ObjAttrib[Vehicle, licensePlate, type_String]
  ObjAttrib[Vehicle, regDate, type_Date]
  ObjFNames[Vehicle,  licensePlate +  regDate +  none]
  // Definition of class Company
  ObjFNames[Company,  cars +  emps +  none]
  // Definition of class Employee
  ObjFNames[Employee,  ins +  none]
  // Definition of class Car
  ObjAttrib[Car, licensePlate, type_String]
  ObjAttrib[Car, regDate, type_Date]
  ObjFNames[Car,  licensePlate +  regDate +  drivenBy +  none]
  // Definition of class Insurance
  ObjAttrib[Insurance, kind, InsuranceKindEnumCDcd1]
  ObjFNames[Insurance,  kind +  none]
  // Definition of class License
  ObjFNames[License,  owner +  none]
  // Definition of class Driver
  ObjAttrib[Driver, exp, DrivingExpEnumCDcd1]
  ObjFNames[Driver,  exp +  license +  drives +  ins +  none]
  // Definition of class Truck
  ObjAttrib[Truck, licensePlate, type_String]
  ObjAttrib[Truck, regDate, type_Date]
  ObjFNames[Truck,  licensePlate +  regDate +  none]

  // Special properties of singletons, abstract classes and interfaces
  no Vehicle

  // Associations
  BidiAssoc[DriverSubsCDcd1, license, LicenseSubsCDcd1, owner]
  ObjLUAttrib[LicenseSubsCDcd1, owner, DriverSubsCDcd1, 1, 1]
  ObjLUAttrib[DriverSubsCDcd1, license, LicenseSubsCDcd1, 0, 3]
  ObjLAttrib[CompanySubsCDcd1, cars, CarSubsCDcd1, 0]
  ObjLU[CarSubsCDcd1, cars, CompanySubsCDcd1, 0, 1]
  ObjLUAttrib[EmployeeSubsCDcd1, ins, InsuranceSubsCDcd1, 1, 1]
  ObjLU[InsuranceSubsCDcd1, ins, EmployeeSubsCDcd1, 1, 1]
  BidiAssoc[DriverSubsCDcd1, drives, CarSubsCDcd1, drivenBy]
  ObjLUAttrib[CarSubsCDcd1, drivenBy, DriverSubsCDcd1, 1, 1]
  ObjLUAttrib[DriverSubsCDcd1, drives, CarSubsCDcd1, 0, 1]
  ObjLAttrib[CompanySubsCDcd1, emps, EmployeeSubsCDcd1, 0]
  ObjL[EmployeeSubsCDcd1, emps, CompanySubsCDcd1, 0]
  // Compositions
```

```
  Composition[InsuranceCompFieldsCDcd1, InsuranceSubsCDcd1]

  Obj = (Vehicle + Company + Employee + Car + Insurance + License + Driver +
      Truck)
}


pred cd {
  cd1
}

// Run commands

run cd for 10 but 5 int
```

Listing A.1: Complete translation result resulting from translating the class diagram given in Listing 2.2. Figure 2.1 depicts the graphical representation of the class diagram.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results

2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07	Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08	Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09	Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11	Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12	Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13	Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14	Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 *	Fachgruppe Informatik: Annual Report 2016
2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud
2016-07	Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.