Modeling transformations using QVT Operational Mappings

P.J.Barendrecht

SE 420606

Research project report

Supervisor: prof.dr.ir. J.E. Rooda
Coaches: dr.ir. R.R.H. Schiffelers
dr.ir. D.A. van Beek

Eindhoven University of Technology
Department of Mechanical Engineering
Systems Engineering Group

Eindhoven, April 2010

Summary

Models play an increasingly important role in the development of (software) systems. The methodology to develop systems using models is called *Model-Driven Engineering* (MDE). One of the key aspects of MDE is transforming models to other models. These transformations are specified in the problem domain. Unlike coding a transformation in the implementation domain, MDE uses different abstraction levels to model a transformation. The transformation is based on the languages of the source- and target models. In the problem domain, such a language is usually called a *meta-model*. A model is an instance of a meta-model when it conforms to the specifications of the meta-model. Finally, the meta-model itself conforms to a higher abstraction level, the *metameta-model*.

With the help of the metameta-model *Ecore*, simple meta-models are designed to experiment with the meta-model of the transformation, the Domain Specific Language *Query/View/Transformation* (QVT) specified by OMG. In other words, QVT is the language used to model transformations. The syntax and semantics of the QVT variant Operational Mappings, often named QVTo, are investigated in this report. The different aspects of the language are explained and illustrated by examples. After the explanation of QVTo, this knowledge is applied to solve a more difficult transformation. The purpose of this transformation, which is in fact the case study, is to transform *CIF* models to *Ariadne* models. The particularities of this transformation are elucidated, such that the reader is able to understand the complete transformation which is included in the appendix.

The models and meta-models are developed in an environment called the *Eclipse Modeling Framework* (EMF). The tools for developing and executing transformations are part of a subproject of the EMF called the *Eclipse M2M project*. This is one of the only two available implementations of QVTo. Unfortunately, this implementation is not completely finished and contains some bugs. In addition, the QVT specifications are not always clear or as practical as they could be. The available documentation is a bit scarce and often not intended for beginners. Because of these shortcomings it can be difficult to model an optimal transformation, however the environment is certainly suitable to model easy to average transformations. It is very likely the environment will be able to manage more complex transformations in the near future, because it has a high potential.

Contents

Chapter 1

Introduction

Models play an increasingly important role in the development of software systems. The methodology to develop systems using models is called Model-Driven Engineering, often abbreviated to MDE. In MDE, models are a substantial part of the solution to a problem, unlike methodologies on a lower abstraction level which use algorithmic concepts to solve problems and use models merely as illustrations.

A model, or abstraction, is an often simplified view of something real, like a machine. To define a model, a modeling language is required. One of the most used modeling notations in software engineering is UML [1], the Unified Modeling Language. However, the language might be too expansive to define a model. Instead of UML, a DSL (Domain-Specific Language) can be used. The syntax of such a language is a model itself, and thus can be seen as a model of the model: a meta-model. When a model adheres to the specifications of the meta-model, it is said to conform to the meta-model. The model is an instance of the meta-model. The meta-model in turn conforms to a model on a higher level, called the metameta-model. There are many ways to model a machine, such that each model describes the same machine. Depending on the objective it could be better to use one specific model instead of another. The same paradigm is valid for meta-models.

In this context it would be very useful to be able to transform (meta-)models to other (meta-)models more suitable for a particular case. Because MDE uses models at different levels of abstraction, such a transformation can be accomplished by using a transformation language, which is a DSL. Aside from model-to-model transformations there are other transformations such as model-to-text and text-to-model transformations.

The main purpose of this report is modeling a transformation from model A to model B, using a DSL. This principle is summarized in Figure 1.1. It shows three models M_{a}, M_{b} and M_{t} (the transformation), each conforming to their own meta-model, respectively MM_{a}, MM_{b} and MM_{t} (the transformation language). M_{a} is the available model and by using M_{t}, which is defined in terms of MM_{a}, MM_{b}, MM_{t}, this model is transformed to M_{b}. MMM represents the metameta-model.

This report is organized as follows. Chapter 2 presents the used modeling languages on the three levels, Chapter 3 explains the transformation language in detail, Chapter 4 illustrates the case study and finally Chapter 5 evaluates the project.

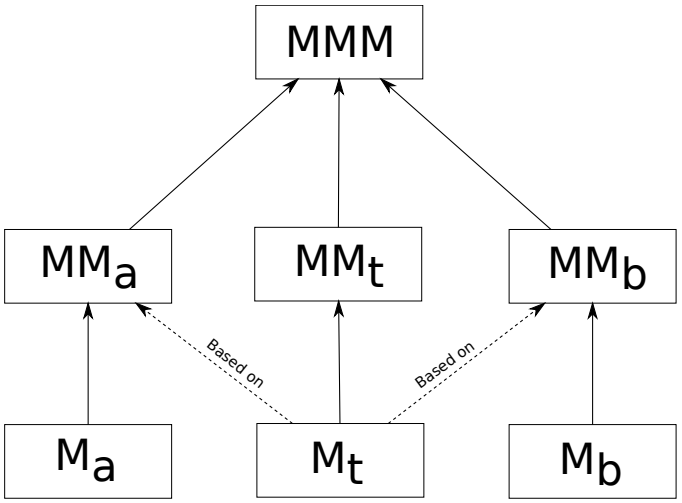

Figure 1.1: Overview of the models, meta-models and metameta-model.

Chapter 2

Instantiaton of the modeling languages

In the introduction, Figure 1.1 represented a general overview of models, meta-models and metameta-models. In this chapter the concrete modeling languages are introduced. Figure 2.1 represents these languages as an instantiation of Figure 1.1. The mentioned models M_1 and M_2 are automata models. The concept of an automaton is briefly discussed in Section 2.1. Ecore, the metameta-model of the automata and the transformation, is discussed in Section 2.2. Two different and simplified meta-models of the automata, MM_1 and MM_2, are discussed in Section 2.3. The language Ecore is used to define these meta-models. The goal of these meta-models is to model a transformation between their instances. This transformation can be found in Appendix B.1 and is used in Chapter 3 to explain the syntax and semantics of QVT Operational Mappings. Finally, the meta-model of the transformation, QVT, is discussed in Section 2.4.

2.1 Automata

An automaton can be used to model for instance a machine. A machine can be seen as a finite amount of states. These states are connected with each other by transitions. A transition connects one source state to one target state. Transitions are triggered by events, i.e. circumstances that cause a transition within the automaton.

An example of a simple automaton representing a valve of a watertank could look like Figure 2.2. This automaton consists of four states which denote the current situation of the valve. The state "Open" represents the situation when the valve is completely opened, while "Closing" represents all positions of the valve between completely opened and closed. The states are connected with each other by six transitions. When "Open" is the current state and the valve is starting to close, the transition from "Open" to "Closing" is triggered by the event "Start_Closing". This same event could trigger

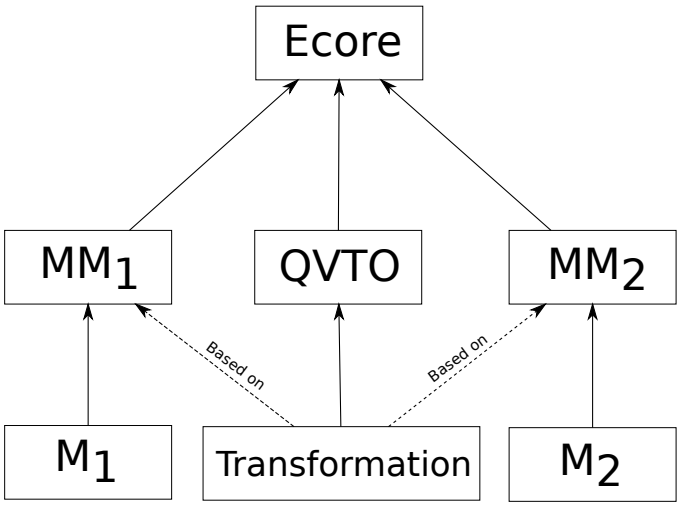

Figure 2.1: Relations between models, meta-models and metameta-model.

the transition from "Opening" to "Closing", but only if the current state would be "Opening".

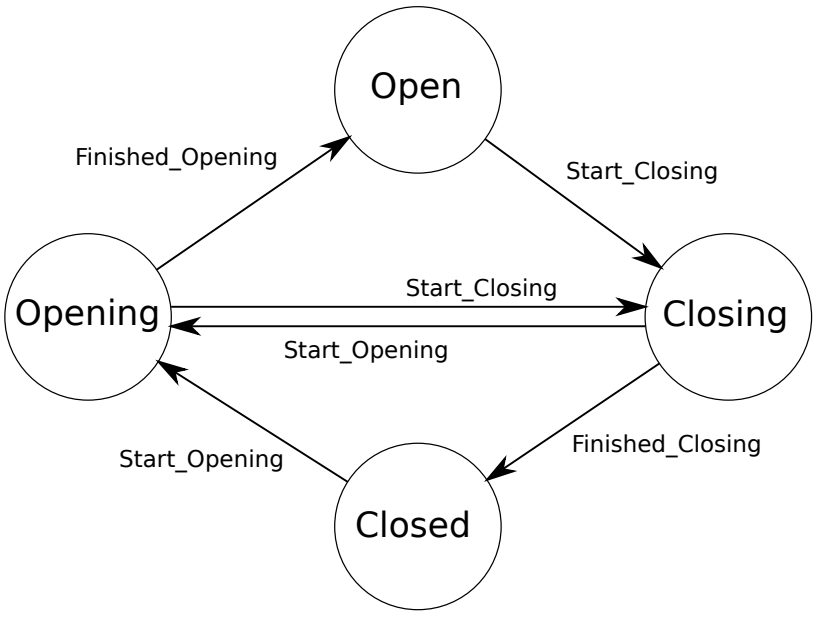

Figure 2.2: An automaton representing a valve of a watertank.

2.2 Ecore, the automata metameta-model

Ecore is part of the EMF, the Eclipse Modeling Framework [2]. Ecore is a graphical Domain Specific Language used to represent models, like UML, the Unified Modeling Language. However, Ecore is a subset of UML. Because Ecore is a model itself, it is its own meta-model and thus there are no higher abstraction levels. Only a few parts of Ecore are used in this report. This subset is shown in Figure 2.3. The diagram shows four classes: EClass, EAttribute, EDatatype and EReference.

EClass represents a modeled class. An EClass has a *name*, zero or more *references* and zero or more *attributes*. An **EReference** is an one-way assiciation between two EClasses: a source and a *target* EClass. The graphical representation is an arrow. An EReference has a *name*, and a *lower*- and *upper bound* used to define the cardinality constraints. These indicate the number of target EClasses the source EClass refers to. The bounds are separated by two dots, where the part on the left presents the lower bound and the right part the upper bound. When the upper bound is unlimited, an asterisk (*) is used. When the lower- and upper bound are equal, they are visualized as a single number. Finally, the Boolean flag *isContainment* indicates whether the target EClasses are contained in the source EClass or not. The graphical representation of a containment is a black diamond. The **EAttribute** represents a modeled attribute. It has again a *name*, a *lower bound* and an *upper bound*. Furthermore the EAttribute conforms to a specific type, an **EDataType**. This can be a basic type like String or Integer, or an object type (wrapped basic types) like a Date.

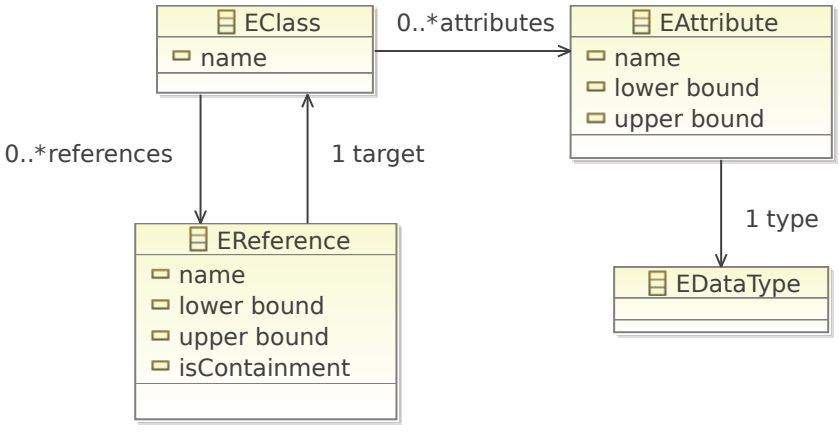

Figure 2.3: Diagram of the subset of Ecore used in this report.

2.3 The automata meta-models

In order to model automata, two meta-models are defined. These two meta-models are called MM_1 and MM_2 (see Figures 2.4 and 2.5). The meta-models conform to Ecore, which is discussed in Section 2.3.

In MM_1, the edges (transitions) are contained within the locations (states). Therefore, the sourceLocation of an edge is implicitly known: it is its parent. The targetLocation is a reference to a location object, this could be the same location as the sourceLocation.

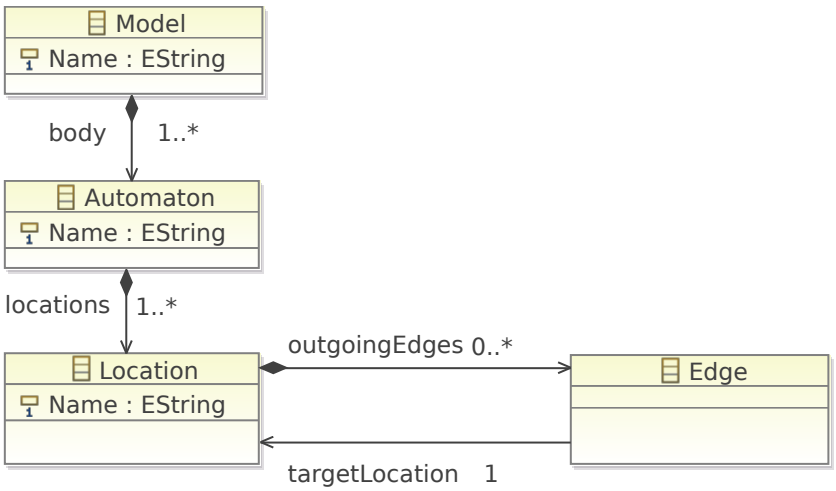

Figure 2.4: Diagram of MM_1.

In MM_2, the edges are contained within the automaton. The sourceLocation is therefore not directly derivable like in MM_1, but now a reference to a location. The targetLocation is a reference to a location as well.

2.4 QVT, the transformation meta-model

The language used for transforming models is QVT, the abbreviation of Query, View, Transformation. It is defined [3] by the OMG, the Object Management Group [4]. The name of the language suggests a three-part structure. The first part is named **Query** because queries can be applied to a source model, an instance of the source meta-model. Next, the **View** is a description how the target model should look like. Finally the **Transformation** is the part where the results of the queries are projected on the view, and thereby creating the target model.

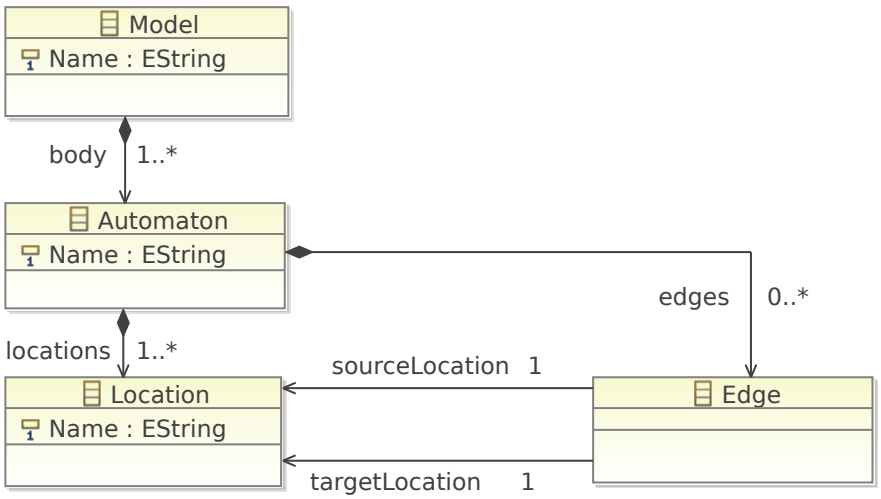

Figure 2.5: Diagram of MM_2.

The specification of QVT depends on the MOF [1][6] and the OCL [7] specifications. OCL, the Object Constraint Language is a formal language originally used to describe constraints in UML models. Nowadays it can be used with any MOF meta-model. OCL can be used for purposes such as specifying pre- and conditions of methods (see Section 3.3.2) and as query language (Section 3.6). QVT consists of three different domain-specific languages: Relations, Core and Operational Mappings. The former two are declarative (a description *what* to investigate, the implementation is left to an interpreter), while the latter is imperative (a direct implementation of transformation instructions). Each of them can be combined with black-box operations (see Section 2.4.4). An overview of the different domain-specific languages is depicted in Figure 2.6. Each language is ultimately transformed [8] to the Core language.

2.4.1 Relations

In the Relations variant a transformation between candidate models (instances of meta-models) consists of a set of relations that should hold between source and target candidate models [9]. Such a relation is defined by two (or more) domains and a couple of pre- and post conditions. A domain is an element of a specific type which can be found in the model. A transformation can be either "checkonly" or "enforced". The former only checks for consistency while the latter enforces consistency by modifying the target model. When the "enforced" option is chosen, the first step is to select one

[1]QVT can be defined in the MOF variant EMOF [3], therefore it can be defined in Ecore as well because EMOF closesly resembles Ecore [5]. This is the reason why in Figure 2.1 Ecore is presented as the metameta-model of the transformation.

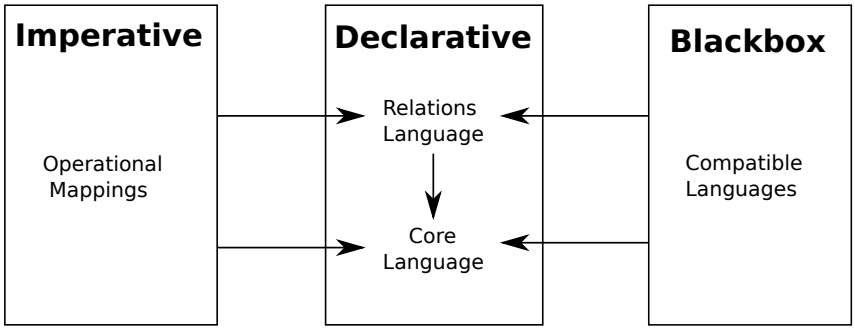

Figure 2.6: Overview of QVT's domain specific languages.

of the candidate models as target. This is not trivial because the defined set of relations is bi-directional: either candidate model can be the source or target. Next the relations are checked. When a relation does not hold, the target model is modified by looking at the domains in the relation. A domain has a pattern, a set of properties and conditions. These are applied to the instance of the domain, i.e. an object. It does not matter whether the target model is empty or already contains model elements.

2.4.2 Core

Core is a small language equally powerful compared to the Relations language. Unlike Relations, Core does not use patterns. Because of its relative simplicity, its semantics can be defined more simply. It is on a lower level than Relations. A transformation in the Relations variant can be transformed [8] to one in the Core variant.

2.4.3 Operational Mappings

Operational Mappings, often called QVTo, is an imperative language that extends Relations and Core. The syntax looks similar to other imperative languages, such as Java. Unlike the declarative variants, Operational mappings is uni-directional: there is a source model and one or more target models. Operational mappings are either transformed [8] to the Relations language and then to the Core language, or directly to the Core language.

2.4.4 Blackbox implementation

A Blackbox allows complex algorithms to be coded in any supported programming language [3]. This is very useful because some algorithms are very hard to implement

in OCL, or cannot be expressed at all. Because OCL is not a programming language, it is not possible to use program logic or use non-query operations. For example, it is not possible to retrieve the current date by using OCL.

2.5 Choosing a domain-specific language

Currently, Operational Mappings is the best supported variant in terms of tools. Although there exist implementations for the Relations variant (MediniQVT [10], ModelMorf [11]), QVTo has a better documentation and a larger user community.

There are two implementations for QVTo: Eclipse M2M [12] and SmartQVT [13]. The first is used in this report. However in retrospect it is regrettable there was not enough time to investigate the other implementation, because seemingly it also is a very suitable implementation.

Chapter 3

QVT Language

In this chapter, the syntax and semantics [3][8] of QVT Operational are explained. In order to present it in a less abstract way, several examples are provided. Most examples originate from the MM_1 to MM_2 transformation, the others are part of a transformation explained on the spot or part of the case transformation, CIFtoAriadne (see Section 4.4). Often used OCL [7] functions are introduced and explained in the examples. To improve the readability of the snippets of code, a similar syntax highlighting is used as in the Eclipse QVT Editor.

3.1 Basic operators, constructions and functions

The operators used in QVT are very similar to the ones from other programming languages. Table 3.1 gives an overview of the most important operators.

Basic constructions to describe the program flow include *if/else* statements, *switch/case* statements, and *while-loops*. It also possible to use *for-loops*. These loops are implemented in a bit unusal way, an example can be found in Section 3.5. The if-statement

Table 3.1: Operators

Description	**Operator**
End of an expression	;
Object and Set Assignment	:=
Set Addition	+=
Equal To	=
Not Equal To	<>
Dot Accessor	.
Arrow Accessor	->

checks whether a boolean condition is true or false. If the condition is true, the statement after "then" is executed, otherwise the statement after "else" is executed.

```
if (condition) then {
  /* Statement 1 */
} else {
  /* Statement 2 */
} endif;
```

It is possible to use conditional expressions in QVT. A conditional expression is a very short way to assign values to an object, depending on a single condition.

```
var a: String;
var b: Integer;
a := ( if (b=b) then "Logic" else "Miracle" endif; )
```

The switch-statement can replace large if/else statements. Each case represents an unique condition. When a case is true, the next statement is executed. When none of the cases are true, the default statement after "else" is executed.

```
switch {
    case (condition1) /* Statement 1 */;
    case (condition2) /* Statement 2 */;
    else /* Statement 3 */;
};
```

Finally, the while-statement or while-loop executes a statement while a condition is true. The condition often includes a count variable whose value is altered in the statement. A while-loop can for instance be used to iterate over a set. It is possible to exit the loop before the condition does not hold anymore. This can be accomplished by using the keyword "break". The other keyword often used in loops, "continue", can be used to skip part of the statement and immediately jump to the start of the statement.

```
while (condition) {
  /* Statement */
};
```

Two important statements used when debugging the transformation are the "log" function and the "assert" expression. The log function displays a message in the terminal, and can be extended with a condition, using the keyword "when". Besides simple text it is possible to display current values of variables. These values can be transformed to the String type either by using the function *repr()*, or *toString()* when it is available. There is a subtle difference between the two: when *repr()* is for example applied

to a floating-point type variable, the represented precission can vary with the *toString()*.

```
log( "message" + variable.repr() + "message" + variable.toString() );
```

The assert function also displays messages in the terminal, but unlike the informative nature of log, it displays warnings and errors. There are three different assertion levels: "warning", "error" or "fatal". When the level is "fatal", the transformation is terminated at that point. The former two just display the message after which the transformation continues.

```
assert level (condition) with log( "message" );
```

3.2 Framework

A QVT Transformation has a certain structure, called the framework. It consists of modeltype definitions, a transformation declaration and a main function.

3.2.1 Modeltype definitions

A modeltype definition is a reference to a meta-model. It it possible to include entire meta-models in the transformation (in-line definition), or reference to externally defined ones. Obligatory parts of a modeltype definition are a name and a reference. This reference can be location specific, i.e. pointing at a file within the Eclipse workspace. In Eclipse this is done by preceding the location with "platform:/resource/".

```
--Modeltype definition using the location specific reference
modeltype MM1 uses "platform:/resource/MM1ToMM2/transforms/MM1.ecore";
```

Additionally the reference can be a package namespace URI, Uniform Resource Identifier. In order to achieve this, the meta-models are registered in Eclipse, after which they can be referenced in the following way.

```
--Modeltype definition using the package namespace URI reference
modeltype MM1 uses "http://mm1/1.0";
```

Apart from these name and reference attributes, a modeltype defnition can be strict (only accepting literal instances of the meta-model) or effective (default, also accepting instances that do not exactly conform to the meta-model). When the definition is strict and the source model is not an instance of the source meta-model, it is not accepted

and there is no transformation at all. When the compliance is effective, only the classes and attributes defined in the meta-model are transformed. To what extend a model can be called "effective" is not explained in the specification. Finally constraints can be imposed using the so-called "where" statement. These constrains are defined using OCL. The current implementation does not support this function, however it is planned.

```
--Adding constraints to the defintion. In this case, the used instance of the meta-model
--should contain at least one Automaton.
modeltype MM1 uses "http://mm1/1.0" where
{ self.objectsOfType(Automaton)->size() >= 1 };
```

3.2.2 Transformation declaration

A transformation declaration consists of a name, an input and an output meta-model. The latter two refer to the defined modeltypes. It is possible to define a transformation which uses multiple input and output meta-models. Additionally a transformation can be extended with or accessed by existing transformations by using the keywords **extends** and **access**. When a transformation is extended, the current transformation inherits all mappings and queries. They can be redefined in the transformation. On the other hand, an accessed transformation is used as a whole (i.e. no individual mappings). It is not possible to modify parts of an accessed transformation. The keywords **new** and **transform** are used to instantiate the accessed transformation (with a copy of the source model), and execute it.

```
transformation MM1ToMM2(in Source:MM1, out Target: MM2);
```

3.2.3 Main function

The actual transformation starts with the **main()** function. The purpose of this function is to set environment variables and call the first mapping. The names of the meta-models can be used in the main function.

```
--Source represents the instance of the source meta-model. Source is a set of all
--elements. By using rootObjects, only the objects at the highest level are selected.
--Additionally, by applying the filter "[Model]" only the Model objects are selected.
main() {
  Source.rootObjects()[Model]->map toModel();
}
```

3.3 Mapping

Mappings form the core of QVT transformations. They ensure that an object from an instance of the source meta-model is transformed into an object in the created instance of the target meta-model. A mapping is invoked by using the keyword **map** or **xmap**. When using **map** and the mapping fails, it returns the **null**-value. When using **xmap**, an exception is raised instead of returning a **null**-value. However, in the current implementation exceptions are not raised (See Eclipse's Bugzilla report #301134).

3.3.1 Declaration

A mapping declaration consists of the classname of the object being transformed, a name of the mapping and the classname of the object that will be the result of the mapping. It it also possible for a mapping to yield multiple target objects.

The name of a mapping should be unique, therefore it is not possible to implement features like polymorphism. A classic example of polymorphism is addition for numbers. When considering two numerical types like Integer and Real, there would be three combinations of two numbers: both Integer, both Real, and twice the possibility one of them is an Integer and the other a Real number. Because the input and output types of a mapping are fixed, it would be useful to create three different mappings to process addition. This is however not possible because of the restriction that a name should be unique. This simple case could be solved in QVT by using disjuncts (see Section 3.6.1), however the plus sign is also often used to concatenate strings. It might not be possible to specify such broad "polymorphism" in QVT.

```
--Mapping declaration of "toModel()". Because both the input and output class is named
--"Model", --it has to be prefixed with the modeltype the objects belong to.
mapping MM1::Model::toModel() : MM2::Model { ... }
```

3.3.2 Conditions

A mapping declaration can be extended with conditions. To execute a mapping, the source object must conform to these conditions, defined using OCL. There are two types of conditions, the pre- and post-condition, indicated with the keywords **when** and **where** respectively. A conversation with a co-developer of the Eclipse M2M implementation revealed there is currently no real use for the post-condition, however it is already implemented for future use.

```
--Pre-condition: the Name of the Model object from the MM1 instance should
--start with an "M"
mapping MM1::Model::toModel() : MM2::Model
```

```
--The keyword "self" refers to the input object.
when { self.Name.startsWith("M"); }
{ ... }
```

3.3.3 Body

The body of a mapping consists of three sections: **init**, **population** and **end**. The purpose of the init-section is to initialize variables and parameters, and print messages to the terminal. The actual mapping is specified in the population-section. Finally, the end-section is intended to contain additional code that should be executed before leaving the mapping.

```
init { log("Arrived in mapping toModel().");  }

--Population section. This keyword can be ommitted.

--Copy the name from the input object to the output object
Name := self.Name;
--A model contains at least one Automaton, each one must be
--mapped after which it is assigned to the target model.
body += self.body->map toBody();

end { log("Leaving the mapping toModel().");  }
```

3.3.4 Parameters

It is possible parametrize a mapping. Parameters should have an unique name (within a specific mapping), and are of a certain type: a basic type like Integer, or an object (possibly declared in the meta-model). Parameters are separated by commas.

```
mapping MM1::Model::toModel( Prefix: String, Loc: MM1::Location ) : MM2::Model
```

In case there are a parameter and a global parameter (see Section 3.8) with the same name, the global one should be accessed by prefixing it with the keyword **this**. The keyword refers to the actual transformation module.

3.4 Resolving

When an object of the source model is transformed to an object in the target model, and later on the same source object appears as a reference, it should not be transformed again. Instead, it should be *resolved* to the object just created in the target model.

Table 3.2: Resolve functions

Resolve Function	Description
resolve	Returns a set of target objects. These objects are the result of an earlier mapping from the source object on which the resolving is being applied.
resolveone	Returns one target object. If the mapping of the source object resulted in multiple target object, only the last is returned.
resolveIn	Like **resolve**, however only the target objects that were mapped in a specific mapping are returned. The name of the mapping is specified by a parameter.
resolveoneIn	Like **resolveIn**, one target object created in a specific mapping is returned.

A simple example is the transformation of a Location from an instance of the MM_1 meta-model (see Figure 2.4) to a Location in an instance of the MM_2 meta-model (see Figure 2.5). When the locations are transformed, the edges still have to be transformed. Each Edge has a reference *targetLocation* which points to one of the just transformed locations. Instead of transforming the location again, it must be resolved to the already transformed Location in the target model. There are different ways to resolve an object.

3.4.1 Source to Target

When a source object is resolved, the resolve-function returns the target objects mapped from this source object. OCL can be used to extend resolvings, i.e. by using conditions or adjusting results. The four resolving variants are concisely explained in Table 3.2.

```
--container() returns the Parent of the source-object as a general object. Therefore
--oclAsType() must be used, after which the location can be resolved to a MM2 Location.
sourceLocation := self.container().oclAsType(MM1::Location).resolveone(MM2::Location);
```

```
--Return the last object that was transformed in the mapping toLocation().
targetLocation := self.targetLocation.resolveoneIn(MM1::Location::toLocation, MM2::Location);
```

```
--Return objects which name starts with an "L", and pick the first result.
targetLocation := self.targetLocation.resolve(m:MM2::Location | m.Name.startsWith("L"))->at(1);
```

3.4.2 Target to Source

These resolvings work in a way identical to the source to target resolvings, but reversed. When a target object is resolved, all source objects are returned. To use either one

of them, the resolve function has to be prefixed with **inv**, resulting in **invresolve**, **invresolveone**, **invresolveIn** and **invresolveoneIn**.

3.4.3 Special cases

When a reference is transformed before the actual object it refers to is transformed, there would be a problem because the reference would be empty. To prevent this, the **late** operator can be added to the resolve statement. The QVT interpreter remembers the position where this operator is used, and fixes these references at the end of the transformation.

```
--The edges were transformed before the locations, so "late" should be added.
targetLocation := self.targetLocation.late resolveone(MM2::Location);
```

3.5 Queries

A query is an operation which uses expressions to obtain data from the input object(s), after which it returns an output object. So far is does not differ from a mapping. However there is a clear difference. Mosty both the input and output objects of a query belong to the same model, whereas the input and output objects of a mapping often belong to different models, i.e. the source and target models. A query can be used like a mapping, however this is not the intention of this operation. Like a mapping, a query can be parametrized. Each parameter can be prefixed with "in" (default), "out" or "inout". When the prefix contains "in", the parameter can be used as input, when it contains "out" as output. However, the current implementation only supports input parameters.

It is possible to call queries from within a blackbox function. In such a case it could be useful to implement a query that prints a message to the terminal, because it may be the case that native display-functions of the used blackbox language are not supported. Two examples of queries that print a message to the terminal can be found below.

```
//...
var Msg: String = "message to be displayed";
Msg.Display();
DisplayParameter(Msg);
//...

query String::Display() {
    --The function firstToUpper changes the first character of the string to a Capital.
    log( self.firstToUpper() );
}
```

```
query DisplayParameter(Message: String) {
    log( Message.firstToUpper() );
}
```

According to the specifications [3] it is not allowed to create new objects within a query, however the Eclipse M2M implemention intentionally allows it. By exploiting this possibility it is possible to return sets of created objects, something which should work fine when using a mapping but currently does not. This concept is illustrated in an example, which is part of the CIF to Ariadne transformation (further discussed in Section 4.4).

```
//...
self.allSubobjectsOfType(Cifx::Update).oclAsType(Cifx::Update) ->
    forEach(Upd) {
        resets += Upd.QueryFromUpdate();
    };
//...

query Cifx::Update::QueryFromUpdate() : OrderedSet(Ariadne::PrimedAssignment) {
    var PrimedAssignments: OrderedSet(Ariadne::PrimedAssignment);

    //...

    --The keyword "return" returns the output object.
    return PrimedAssignments;
}
```

3.6 Inheritance

The transformation of multiple subclasses (inheritted from one superclass, in Ecore graphically represented by an arrow with a white head) to multiple subclasses in the target model could be difficult to accomplish, because the transformation is applied to the superclasses. Figure 3.1 illustrates this problem. There are two simple metamodels MM_α and MM_β, which both represent a collection of shapes or figures. When transforming instances of MM_α to instances of MM_β, the shapes are transformed to figures. However, it is not the Shape class itself that should be transformed, but its subclasses, Ellipse_α and Polygon. Ellipse_α is transformed to Ellipse_β, Polygon (depending on the number of sides) to Triangle or Rectangle. There might be attributes in the superclass (e.g. *Color*) that need to be transformed as well. Most times they can be transformed in the usual way, however sometimes it might be useful to use separate mappings for this purpose. Section 3.6.2 discusses this matter in more detail.

This transformation could be accomplished in several ways. In the first example the subtype of the Shape object is determined in the mapping which transforms the Shapes to Figures.

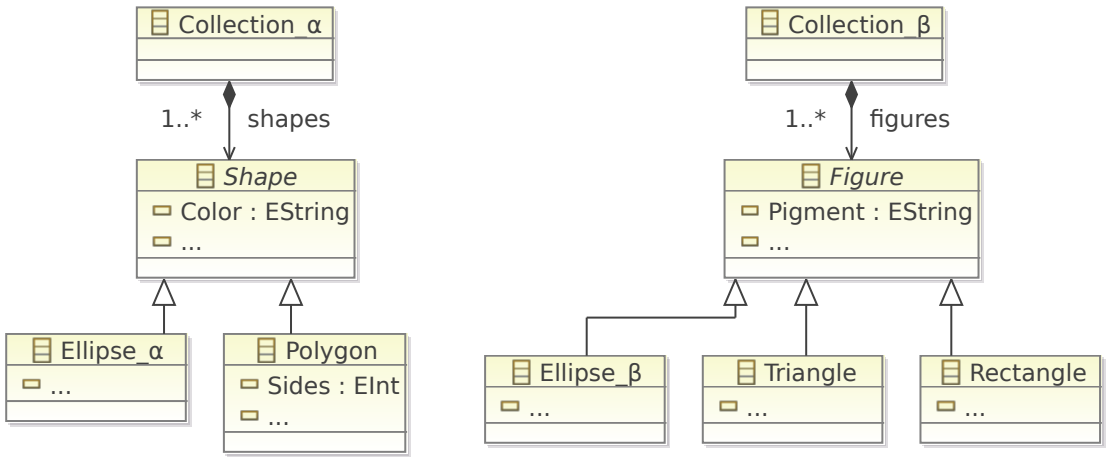

Figure 3.1: The meta-models MM_α (left) and MM_β (right).

```
mapping Collection_α::toCollectionBeta() : Collection_β {
    figures := self.shapes->map toFigures();
}

--The variable "result" is the result type of the mapping, i.e. Figure
--or one of its subtypes, Ellipse_β, Triangle or Rectangle.
mapping Shape::toFigures() : Figure {
    Pigment := self.Color;
    if Shape.oclIsTypeOf(Ellipse_α) then {
        result := object Ellipse_β { /*Transformation of attributes ... */ };
    } else {
        --It is a Polygon|ref{}
        if Shape.oclAsType(Polygon).Sides = 3 then {
            result := object Triangle { /*Transformation of attributes ... */ };
        } else {
            result := object Rectangle { /*Transformation of attributes ...*/ };
        } endif;
    } endif;
}
```

In the second example the subtype of the Shape is determined in the mapping which evokes the actual transformation mappings.

```
mapping Collection_α::toCollectionBeta() : Collection_β {
    --By using the operator "+=", the resulting objects are added to the set,
    --they do not replace the existing data.
    figures += self.shapes->select(s | s.oclIsTypeOf(Ellipse_α)).oclAsType(Ellipse_α)->
        map toEllipseBeta();
    figures += self.shapes->select(s | s.oclIsTypeOf(Polygon) and
```

```
        s.oclAsType(Polygon).Sides=3).oclAsType(Polygon)->map toTriangle();
    figures  += self.shapes->select(s | s.oclIsTypeOf(Polygon) and
        s.oclAsType(Polygon).Sides=4).oclAsType(Polygon)->map toRectangle();
}

mapping Ellipse_α::toEllipseBeta() : Ellipse_β {
    /*Transformation of attributes ... */
}

mapping Polygon::toTriangle() : Triangle {
    /*Transformation of attributes ... */
}

mapping Polygon::toRectangle() : Rectangle {
    /*Transformation of attributes ... */
}
```

Because the size of this example is very small, code still looks organized. However, if there would be more subclasses or conditions, it would get messy. In these cases the **disjuncts** option function could be very useful.

3.6.1 Disjuncts

The keyword **disjuncts** compares the type of the source object (e.g. *Shape*) of a "disjunct mapping" (e.g. *toFigures*) to the types of source objects of the indicated mappings after the keyword **disjuncts**. The first matching mapping performs a transformation on the object. Besides the types of the source objects, the disjunct mapping considers the pre- and post conditions of the indicated mappings. In other words, a disjunct mapping acts like a referrer: it checks the type of the input object and passes it on to the first indicated mapping able to transform the object.

The code below yields the same results as the two earlier snippets in this section, however it looks more well-organized.

```
mapping Collection_α::toCollectionBeta() : Collection_β {
    figures  := self.shapes->map toFigures();
}

mapping Shape::toFigures() : Figure
disjuncts Ellipse_α::toEllipseBeta, Polygon::toTriangle, Polygon::toRectangle
{ /* No mapping body */ }

mapping Ellipse_α::toEllipseBeta() : Ellipse_β {
    /*Transformation of attributes ... */
}

--The disjuncts function in the mapping toFigures() checks the pre-condition.
--Iff the Polygon has 3 sides it is transformed to a Triangle.
```

```
mapping Polygon::toTriangle() : Triangle
when { self.Sides = 3 }
{
    /*Transformation of attributes ... */
}

mapping Polygon::toRectangle() : Rectangle
when { self.Sides = 4 }
{
    /*Transformation of attributes ... */
}
```

3.6.2 Inherits and Merges

Mappings can be reused in other mappings. There exist two options to accomplish this: **merges** and **inherits**. An inherited mapping is executed between the **init** and **population** section of a mapping, a merged mapping after the **end**-section of a mapping. The latter could be used to override results of the mapping itself.

In the example above the Shape class has an attribute Color. The subclasses Ellipse_α and Polygon inherit this attribute. In the code presented in Section 3.6.1 the attribute Color is not yet transformed to Pigment. Unfortunately this can not be done in the mapping *toFigures*, because this is a disjunct mapping which do not use their mapping body. Likewise the **merges** and **inherits** options can not be used in this mapping, because they can not be combined with **disjuncts**. However these options can be used in mappings like *toEllipse_*α. The code below provides a way to use **merges** and **inherits**.

```
mapping Collection_α::toCollectionBeta() : Collection_β {
    figures := self.shapes->map toFigures();
}

mapping Shape::ownAttributes() : Figure {
    Pigment := self.Color;
}

mapping Shape::toFigures() : Figure
disjuncts Ellipse_α::toEllipseBeta, Polygon::toTriangle, Polygon::toRectangle
{ /* No mapping body */ }

mapping Ellipse_α::toEllipseBeta() : Ellipse_β
inherits Shape::ownAttributes {
    /*attributes ... */
}

mapping Polygon::toTriangle() : Triangle
--Although this mapping first assigns the value "Yellow" to the attribute Pigment,
--it is later on replaced by the Color of the Shape by using the merged mapping "ownAttributes".
merges Shape::ownAttributes
```

```
when { self.Sides = 3 }
{
    Pigment := "Yellow";
    /*attributes  ... */
}
```

3.7 Intermediate Classes and Properties

It is possible to define intermediate classes and properties in a transformation. They can be used to make a part of the transformation easier. An intermediate class is either an entirely new class or a class which extends an existing class. The class type can be used anywhere in the transformation. An intermediate property is an attribute which is added to an existing class. The type of an intermediate property is often an intermediate class. Intermediate classes and properties do not appear in the target model, they are temporarily. The syntax used to implement these intermediate classes and properties is the same syntax used to declare in-line meta-models (see Section 3.2.1).

To illustrate this concept, consider the meta-models MM_1 and MM_2. Instead of transforming all Locations, it could be useful not to transform the deadlock locations. A deadlock location is a location which does not have any outgoing edges. In other words, the state of the automaton can not change anymore. To achieve this, an intermediate class "Deadlock" is defined. Each deadlock location will be transformed to a Deadlock object. This class has the attributes Name and originalLocation. The latter is a reference to the original deadlock location, not a containment. Furthermore, the class Automaton is extended with two attributes: deadlocks and nonDeadlocks. To check whether a Location has any outoing edges, the OCL function *size()* is used which returns the size of a list.

```
intermediate class Deadlock {
    Name: String;
    references originalLocation: MM1::Location[1];
}

intermediate property MM1::Automaton::deadlocks : Set(Deadlock);

intermediate property MM1::Automaton::nonDeadlocks : Set(MM1::Location);

//...

mapping MM1::Automaton::toBody() : MM2::Automaton {

    self.locations->forEach(Loc) {
        if ( Loc.outgoingEdges->size() = 0 ) then {
            --A deadlock state, transform Location to Deadlock and add it
            self.deadlocks += object Deadlock {Name := "Deadlock_" +
                                Loc.Name; originalLocation := Loc};
```

```
        } else {
            --Not a deadlock state, add Location to the set  nonDeadlocks
            self.nonDeadlocks += Loc;
        } endif;
    };

    locations  += self.nonDeadlocks.map toLocation();

    //...

}
```

The above example is only intended to illustrate the possibilities of intermediate classes and intermediate properties. Before the mentioned idea would be usable, the edges should be split as well, because an edge pointing towards a deadlock location should not be transformed (it is not possible to resolve a target Location that will not be transformed).

3.8 Parameters (configuration properties)

Within a transformation, global parameters, consisting of a name and type, can be defined by using the keywords **configuration property**. They cannot be initialized in the code itself, only from outside the transformation. This option is available in the Transformation Running Configuration in Eclipse.

This example is part of the CIF to Ariadne transformation, further discussed in Section 4.4.

```
configuration property Prefix: Boolean;
```

3.9 Standalone execution

According to several sources [14][15] it should be possible to execute a transformation from a Java plug-in. It is unclear whether it is possible to write a standalone application, i.e. an application which can be used to execute a finished transformation without the Eclipse environment.

Chapter 4

Case study, CIFtoAriadne

The case itself is about transforming instances of the CIF meta-model to instances of the Ariadne meta-model. First a description of the CIF and Ariadne meta-models is given, after which the transformation is investigated.

4.1 CIF

CIF, the Compositional Interchange Format [16], can be used to model systems such as hybrid automata. A hybrid system is a combination of continuous time evolution and discrete events. To make this possible, a normal automaton (however much more complex than the one from the introduction) is extended with the possibility to associate differential equations with states. CIF supports arbitrary differential-algebraic equations (DAEs), a wide range of urgency concepts, and parallel composition. The purpose of CIF is to achieve inter-operability by means of model transformations to and from the CIF language. Ecore diagrams are used to define the CIF language. The CIF meta-model can be found in Appendix A.1, Figures A.1, A.2 and A.3.

4.2 Ariadne

Ariadne [17] is a library for computation with hybrid automata. The purpose of Ariadne is to support analysis and synthesis of systems described with hybrid automata. To realize this, it is being built to be an open and easily extensible package that features basic data structures and operators. It is implemented in such a way that it is very easy to add new types and algorithms. Similar existing tools are restricted: they often only support timed automata or (limited) hybrid ones, and moreover, they are closed source. Ariadne does not have these limitations.

The name *Ariadne* originates from the name of the daughter of King Minos of Crete, Ariadne [18]. In this myth, she helped Theseus find his way out of the labyrinth. The tool aims to help users find their way out of the hybrid system labyrinth.

Recently, the input language of Ariadne was defined using Ecore class diagrams. Figure A.4 in Appendix A.2. shows the Ariadne meta-model.

4.3 Instance of the CIF meta-model

The modeled transformation from CIF to Ariadne models can be applied to many CIF models. However, in order to comprehend the transformation, it is elucidated using a specific instance of the CIF meta-model. The Eclipse representation of the CIF model is depicted in Figure 4.1. The instance of the Ariadne meta-model (see Section 4.5) is the result of the transformation when using this CIF model.

The CIF model represents a watertank. The representation consists of two automata (more complex than the automata introduced in Section 2.1). One automaton represents the tank, the other automaton the valve. The first automaton consists of just one state (location), the second automaton consists of four states and six edges, like the automaton in Figure 2.2. Each Location can contain multiple flow-, invariant- or tcp Expressions. Each Edge can contain multiple guard Expressions, at most one BaseUpdate and one BaseAction. In Eclipse, child objects are indented with respect to their parents. This means that both automata are children of the object "Parallel Composition", which is a child of "Scope Scope 1". This Scope also contains two Variables, height and alpha, and four Actions. Finally, the grandparent of the Scope, "Specification", contains six Constant Declarations.

4.4 Transformation

In this section, the particularities of the transformation from CIF to Ariadne are discussed. These special parts of the transformation are treated chronological. Before these technical aspects are handled, it is important to realize only a subset of CIF can be transformed to Ariadne. Besides, not this entire subset is considered in the transformation. The actual subset consists of the classes with a distinct black rectangle around them (see Appendix A.1).

4.4.1 Framework

Because the CIF language covers multiple Ecore diagrams, each diagram is declared individually as a modeltype in the transformation. Aside from the main modeltype of the CIF language, two others are declared: *Expressions* and *Type*. These modeltypes

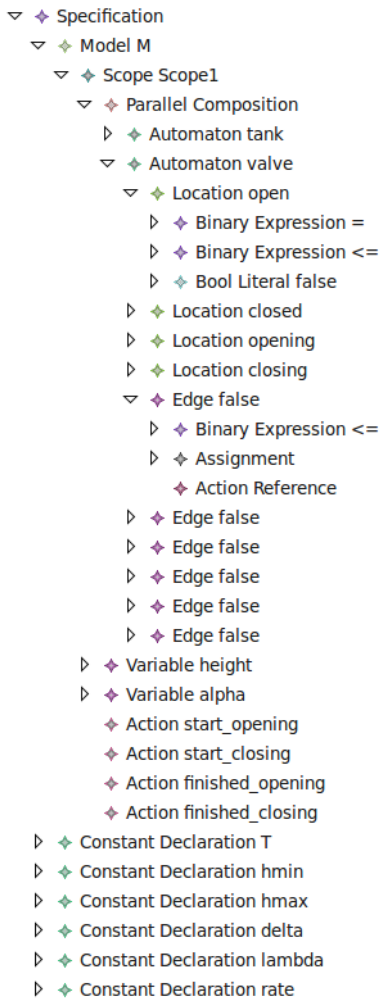

Figure 4.1: The instance of the CIF meta-model

appear as abstract classes in CIF's main modeltype, and link to the actual modeltypes. Because of these links, only the main modeltype is part of the transformation declaration.

```
modeltype Cifx uses "http://se.wtb.tue.nl/cif/cif-2.1.0";
modeltype Expressions uses "http://se.wtb.tue.nl/cif/expressions-2.1.0";
modeltype Type uses "http://se.wtb.tue.nl/cif/types-2.1.0";
modeltype Ariadne uses "http://ariadne.org/ariadne-1.0.0";

transformation CifxToAriadne(in modelIn:Cifx, out modelOut:Ariadne);
```

There is another particularity in the framework of the transformation, a configuration property *Prefix*. In CIF, a Model object can contain multiple scopes (not on the same

level), and these scopes multiple variables. However, in Ariadne all variables are contained in the model. Hence it could be useful to prefix the CIF variables with the name of their parent, a Scope. The boolean configuration property Prefix defines whether the variables are prefixed.

```
configuration property Prefix: Boolean;

//...

--From the toVariable mapping:
if ( self.container().oclIsTypeOf(Scope) and Prefix ) then {
        name := self.container().oclAsType(Scope).name + "_" + self.name;
    } else {
        name := self.name;
    } endif;
```

4.4.2 Mappings

As regards the mappings, there are a few interesting details as well. In CIF, the Constant Declarations are contained within the Specification object, like the Model object. In Ariadne the Constants are contained in the Model object. Therefore the transformation must move the Constants to another level. This is accomplished by using the OCL function *container()*, which returns the parent object of a source object.

```
--From the toModel mapping:
--fromTopLevelDeclaration is a disjunct mapping.
constants += self.container().oclAsType(Cifx::Specification).declarations->
        map fromTopLevelDeclaration();
```

In Ariadne all automata are contained within one Model object. In this perspective, CIF is different because an automaton can be contained within a Scope or ParallelComposition, which can also be contained in another Scope or ParallelComposition. It is possible to implement a recurse mapping which transforms all automata, yet this is not necessary because only the automata themselves have to be transformed. In other words, it does not matter which object is their parent. Therefore, the OCL function *allSubobjectsOfType()* is used to recursively find all automata.

```
automata += self.allSubobjectsOfType(Cifx::Automaton)->oclAsType(Cifx::Automaton).
        map toAutomaton();
```

The transformation of the Expression is next. Many classes in CIF, such as Location, Edge, ConstantDeclaration and AddressableVariable, contain an Expression. Because the Expression class has many subclasses like BinaryExpression or VariableReference, it

is useful to model the Expression mapping a disjunct mapping.

```
mapping Expressions::Expression::toExpression() : Ariadne::Expression
disjuncts Expressions::VariableReference::toExpVar,
          Expressions:: ConstantReference::toExpConst,
          Expressions:: UnaryExpression::toExpUnary,
          Expressions:: BinaryExpression::toExpBinary,
          Expressions:: LiteralExpression:: toLiteral ,
          Expressions:: FunctionCallExpression::toFunction
{}
```

Another difficult case is the mapping of CIF's Update and Assignment classes to Ariadne's PrimedAssignment class. An Edge in CIF can contain at most one Update or Assignment, which in their turn can contain multiple variables and values. In Ariadne it is the other way round. A Transition can contain multiple PrimedAssignments which can only contain one variable and one value. This means that one Update or Assignment object can result in multiple PrimedAssignment objects, and therefore the mapping has to return a set of PrimedAssignments. Because of the bug mentioned earlier on (see Section 3.5), a query is used instead of a mapping. Additionaly, this part of the transformation shows that an Expression can consists of multiple Expressions, another reason why the disjunct mapping is useful.

```
//...
self.allSubobjectsOfType(Cifx::Update).oclAsType(Cifx::Update) ->
    forEach(Upd) {
        resets += Upd.QueryFromUpdate();
    };
//...

query Cifx::Update::QueryFromUpdate() : OrderedSet(Ariadne::PrimedAssignment) {
    var PrimedAssignments: OrderedSet(Ariadne::PrimedAssignment);

    if ( self.variables->size() <> self.expressions->size() ) then {
        log ("QueryFromUpdate: The number of variables does not correspond to the number of
            expressions. This is not supported at the moment.");
    } else {
        self.expressions->forEach(Exp) {
            if ( Exp.oclIsTypeOf(Expressions::BinaryExpression) ) then {
                if ( Exp.oclAsType(Expressions::BinaryExpression).leftChild.oclIsTypeOf
                (Expressions:: VariableReference) ) then {

                    --Create a PrimedAssignment object with attributes primedVariable and value
                    PrimedAssignments += object PrimedAssignment {
                        primedVariable := Exp.oclAsType(Expressions::BinaryExpression).leftChild.
                                oclAsType(Expressions::VariableReference).variable.
                                resolveone(Ariadne::Variable);

                        value := Exp.oclAsType(Expressions::BinaryExpression).rightChild.
                                map toExpression();
```

```
                    };

                } else {
                    log("fromUpdate: Left child is not a VariableReference.");
                } endif;
            } else {
                log("fromUpdate: Expression is not a BinaryExpression.");
            }endif;
        }; --End forEach

    } endif;

    --The keyword "return" returns the output object.
    return PrimedAssignments;
}
```

Finally, there is another disjunct mapping, this time with conditions. Certain Expressions in CIF have to be transformed to Predicates in Ariadne. However, there are two types of Predicates (i.e. BinaryPredicate and ComparisonPredicate) that are transformed from BinaryExpression objects. Because of this reason, the *toBinaryPredicate*- and *toComparisonPredicate* mapping both perform a check on the input object. When the operator of the BinaryExpression is either *GreaterEqual* or *LessEqual*, the object is transformed to a ComparisonPredicate, otherwise a BinaryPredicate is the result.

```
mapping Expressions::Expression::toPredicate() : Ariadne::Predicate
disjuncts Expressions::LiteralExpression:: toConstantPredicate,
          Expressions:: UnaryExpression::toUnaryPredicate,
          Expressions:: BinaryExpression::toBinaryPredicate,
          Expressions:: BinaryExpression::toComparisonPredicate
{}

mapping Expressions::BinaryExpression::toComparisonPredicate() : Ariadne::ComparisonPredicate
when { (self.operator = Expressions::BinaryOperator::GreaterEqual) or (self.operator =
          Expressions:: BinaryOperator::LessEqual) }
{
    //...
}

mapping Expressions::BinaryExpression::toBinaryPredicate() : Ariadne::BinaryPredicate
when { (self.operator <> Expressions::BinaryOperator::GreaterEqual) and (self.operator <>
          Expressions:: BinaryOperator::LessEqual) }
{
    //...
}
```

4.5 Instance of the Ariadne meta-model

After applying the modeled transformation to the instance of the CIF meta-model (see Section 4.3), the resulting model is an instance of the Ariadne meta-model. The Eclipse representation of this instance is depicted in Figure 4.2. A notable difference to the CIF model is the general structure of the model, i.e. in the Ariadne model the object "Model M" is the parent of the Automata, Variables, Events and Named Constants, whilst the same object "Model M" in the CIF model is a child itself.

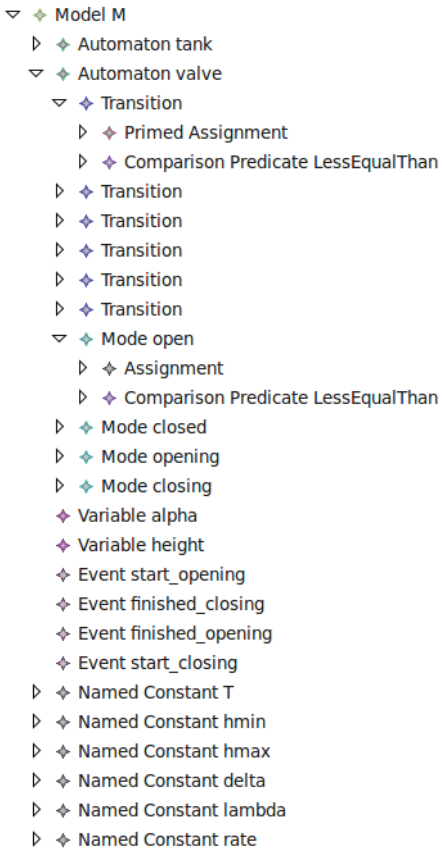

Figure 4.2: The instance of the CIF meta-model

Chapter 5

Evaluation

In this chapter the project is evaluated. First, the conclusions are presented, after which suggestions for future investigation are given.

5.1 Conclusions

The first implementation of QVT Operational Mappings was released in 2007. Since this is only a few years ago, there is not much documentation available yet. The available documentation explains QVT on a rather high level, causing a steep learning curve. The missing link in learning QVT are resources written for the absolute beginner in the model-to-model transformation domain. Even the official MOF QVT specification does not explain the matter clear enough for a beginner to entirely understand it. However, with the help of the QVT user community it was possible to learn QVTo within a relatively short period of time. After experimenting with the QVTo language using the simple meta-models MM1 and MM2 and some other experiments (not part of this report), it is concluded that QVTo might be a suitable language to model the CIF2Ariadne transformation.

During the modeling of the transformations, difficult problems were addressed. Some solutions to these problems are a bit cumbersome, because the Eclipse implementation has not implemented all available QVT features yet, or because the specification is not as practical as it could be (e.g. no mapping body in a disjunct mapping, neither combinable with inherits or merges). Other problems revealed bugs in the implementation which led to bug reports at Eclipse's Bugzilla.

The general conclusion can be split in two parts: the specification of the QVTo language and its implementation. The specification could be more practical and its documentation clearer, especially in view of beginners. However, QVTo is an adequate language to model complicated transformations. The Eclipse M2M implementation of QVT Operational is usable to implement rather simple transformations. It will take some time

before highly complex transformations can be implemented completely, but this will surely happen because the project has a high potential.

5.2 Future work

Because of the current bugs in the Eclipse M2M implementation it would be very useful to investigate the other available implementation for QVT Operational Mappings, SmartQVT. There was too little time left to do this in the current project.

Another possible future investigation results from a short study to the declarative variants. They could have advantages with respect to imperative variants, for example less work to implement the transformations, i.e. no program flow specification required.

Finally, it would be useful to be able to execute the transformation as a stand-alone application, with the possibility to slightly change the implementation of the transformation. This way a finished transformation could be distributed as a piece of software runnable on almost any computer, without the need to install extra software to execute it.

Bibliography

[1] The Object Management Group. *The UML ® Resource Page.* http://www.uml.org/.

[2] The Eclipse Foundation. *The Eclipse Modeling Framework Project.* http://www.eclipse.org/modeling/emf/.

[3] The Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification*, 2007.

[4] *Object Management Group.* http://www.omg.org/.

[5] Dave Steinberg, Frank Budinksy, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework.* Eclipse Series. 2009.

[6] The Object Management Group. *Meta Object Facility (MOF) Core Specification*, 2006.

[7] The Object Management Group. *Object Constraint Language, OMG Available Specification*, 2006.

[8] Siegfried Nolte. *QVT - Operational Mappings, Modellierung mit der Query Views Transformation.* Springer-Verlag, 2010.

[9] Siegfried Nolte. *QVT - Relations Language, Modellierung mit der Query Views Transformation.* Springer-Verlag, 2009.

[10] ikv++ technologies ag. *Medini QVT.* http://projects.ikv.de/qvt.

[11] Tata Research Development and Design Centre. *ModelMorf.* http://121.241.184.234:8000/ModelMorf/ModelMorf.htm, 2007.

[12] The Eclipse Foundation. *Model To Model (M2M).* http://www.eclipse.org/m2m/.

[13] France Telecom R&D. *SmartQVT.* http://smartqvt.elibel.tm.fr/.

[14] The Eclipse Foundation. *An example of launching QVT Operational transformations programatically in Java.* http://wiki.eclipse.org/QVTOML/Examples/InvokeInJava, 2009.

[15] Radomil Dvorak and Sergey Boyko. *Standalone QVTO ececution, Buzilla Bug 287685.* https://bugs.eclipse.org/bugs/show_bug.cgi?id=287685, 2009.

[16] The Systems Engineering Wiki. *General information about the CIF.* http://se.wtb.tue.nl/sewiki/cif/general, 2009.

[17] University of Udine and PARADES and CWI and University of Verona. *Ariadne: An open tool for hybrid system analysis.* http://trac.parades.rm.cnr.it/ariadne/.

[18] *Ariadne as tool's name.* http://trac.parades.rm.cnr.it/ariadne/wiki/faq.

Appendix A

Meta-models

A.1 CIF

Because CIF is a more extensive language than Ariadne, only a part of it can be transformed. Therefore, some classes in this meta-model have a distinct black rectangle around them. Together these classes represent the subset of the CIF language that is transformed to Ariadne in the CIFtoAriadne transformation.

The diagram of the CIF meta-model is splitted into several parts.

A.2 Ariadne

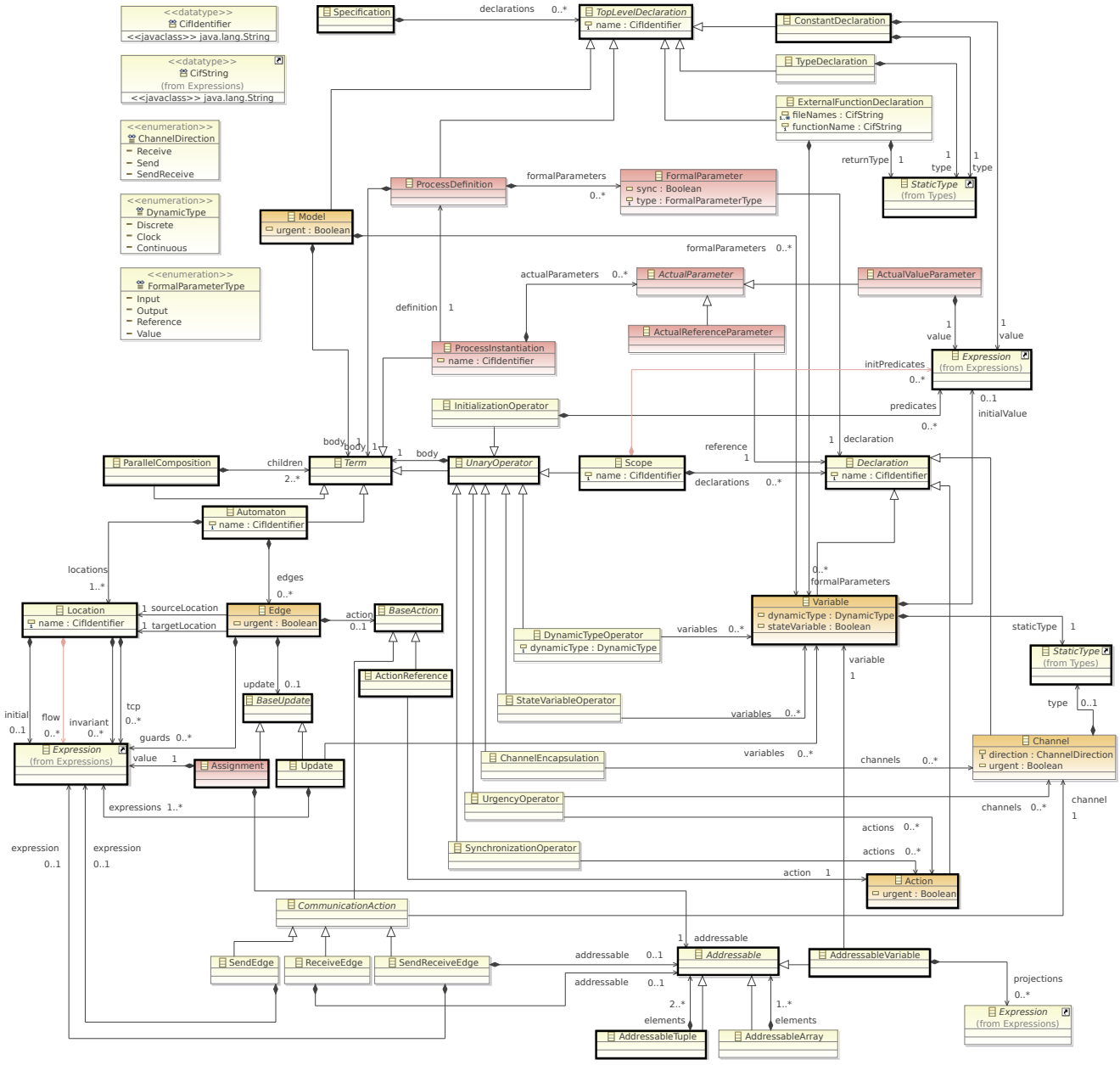

Figure A.1: The main part of the CIF meta-model.

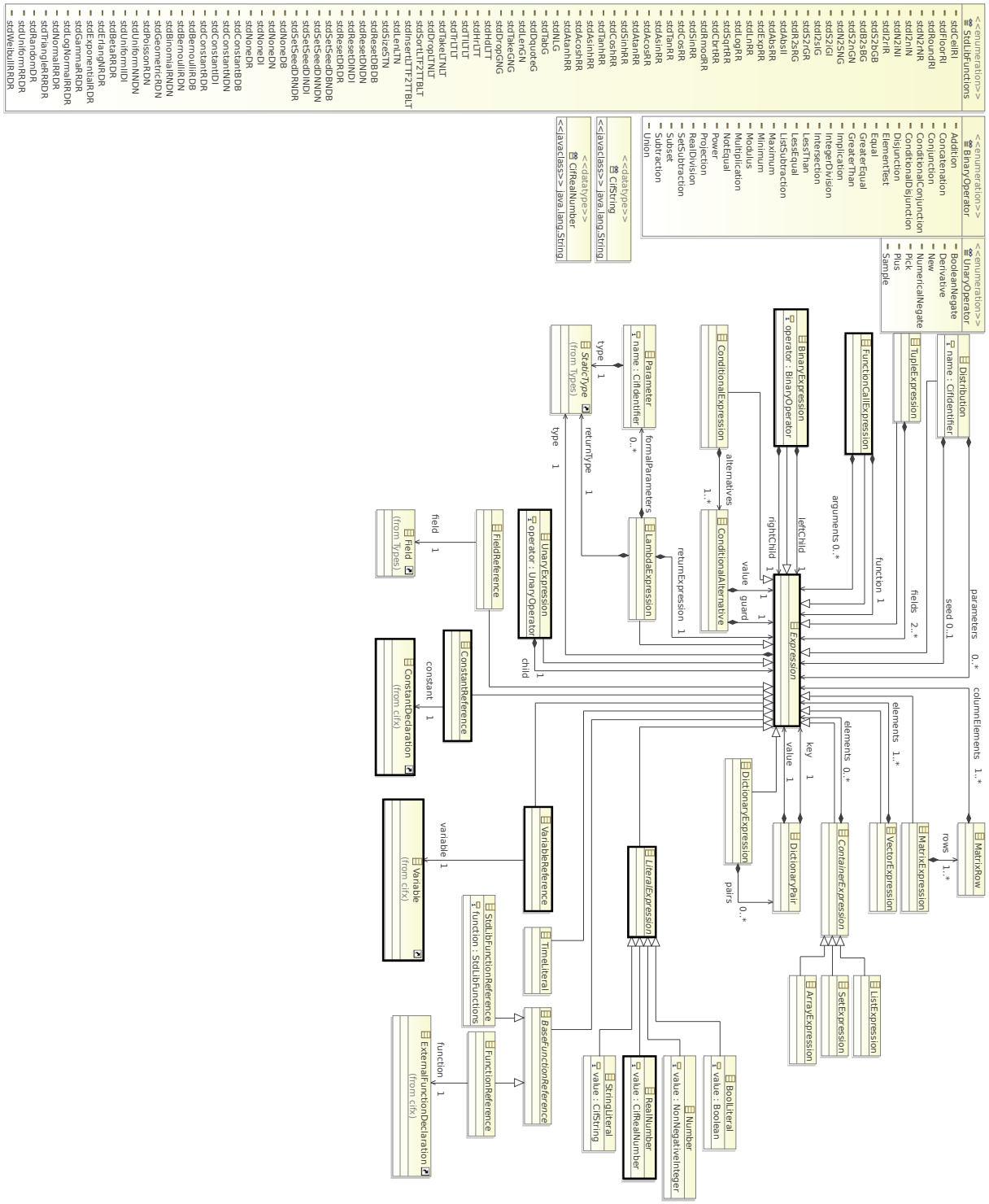

Figure A.2: Part of the CIF meta-model, Expressions.

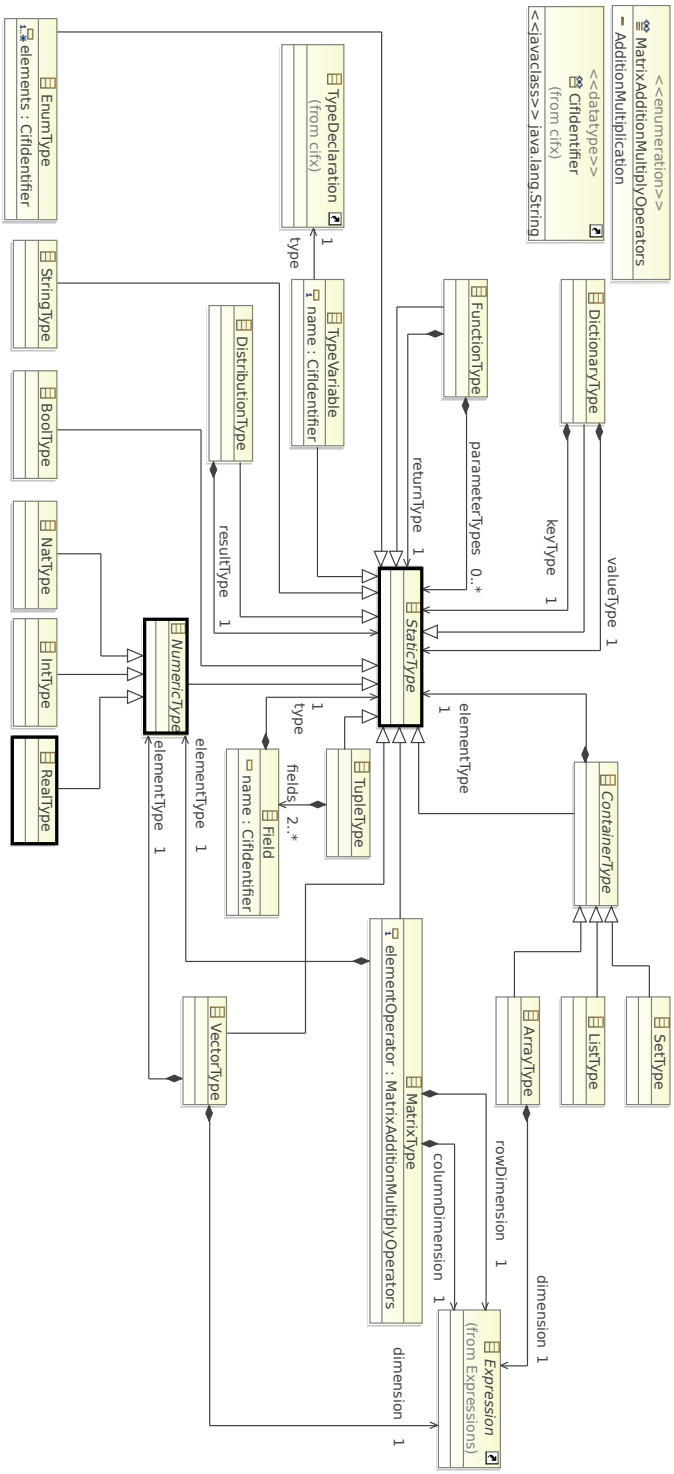

Figure A.3: Part of the CIF meta-model, Types.

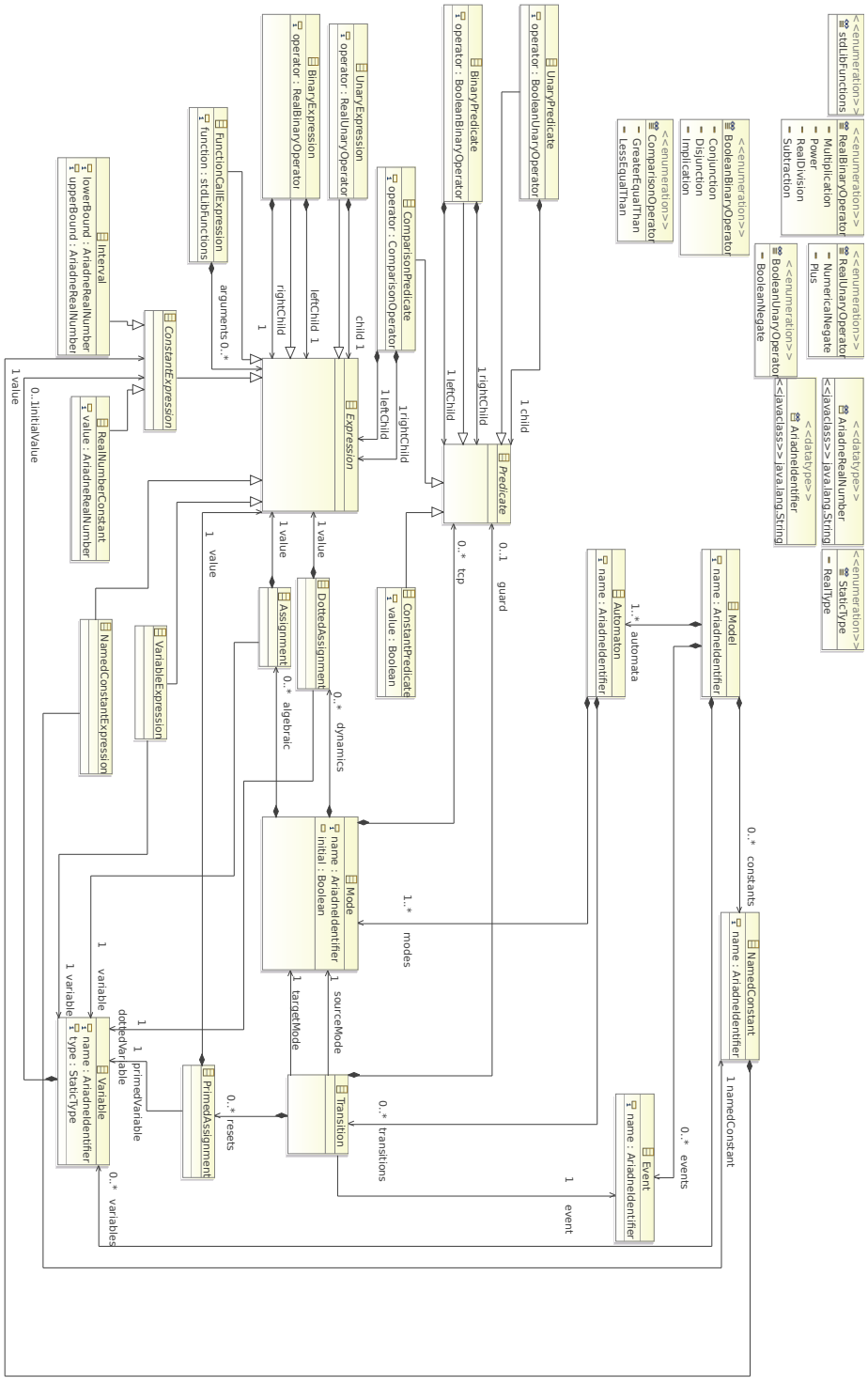

Figure A.4: The Ariadne meta-model.

Appendix B

Transformations

B.1 MM1toMM2

```
modeltype MM1 uses "http://mm1/1.0";
modeltype MM2 uses "http://mm2/1.0";

transformation MM1ToMM2(in Source:MM1, out Target: MM2);

main() {
    --Find the root objects and invoke the mapping to transform them
    Source.rootObjects()[Model]->map toModel();
}

mapping MM1::Model::toModel() : MM2::Model {
    init { log("Arrived in mapping toModel().");}

    --Copy the name from the input object to the output object
    Name := self.Name;
    --A model contains at least one Automaton, each one must be
    --mapped after which it is assigned to the target model.
    body += self.body->map toBody();

    end { log("Leaving the mapping toModel()");}
}

mapping MM1::Automaton::toBody() : MM2::Automaton {
    Name := self.Name;

    locations += self.locations.map toLocation();
    --Transform edges after locations (in connection with resolving)
    edges += self.locations.outgoingEdges.map toEdge();
}

mapping MM1::Location::toLocation() : MM2::Location {
    Name := self.Name;
}

mapping MM1::Edge::toEdge() : MM2::Edge {
    --In MM1 the source location is the parent of the edge,
    --use container to find tihs parent object and resolve it.
    sourceLocation := self.container().oclAsType(MM1::Location).resolveone(MM2::Location);

    --The target location is in both meta-models a reference, so it only
    --needs to be resolved.
    targetLocation := self.targetLocation.resolveone(MM2::Location);
}
```

B.2 CIFtoAriadne

```
modeltype Cifx uses "http://se.wtb.tue.nl/cif/cif-2.1.0";
modeltype Expressions uses "http://se.wtb.tue.nl/cif/expressions-2.1.0";
modeltype Type uses "http://se.wtb.tue.nl/cif/types-2.1.0";
modeltype Ariadne uses "http://ariadne.org/ariadne-1.0.0";

transformation CifxToAriadne(in modelIn:Cifx, out modelOut:Ariadne);

configuration property Prefix: Boolean;

main() {
   modelIn.objects()[Model]->xmap toModel();
}

mapping Model::toModel() : Ariadne::Model {
   name := self.name;

   constants += self.container().oclAsType(Cifx::Specification).declarations->map fromTopLevelDeclaration();

   if ( self._body.oclIsTypeOf(Scope) ) then {

      variables += self._body.oclAsType(Scope).allSubobjectsOfType(Cifx::Variable)->
            oclAsType(Cifx::Variable).map toVariable();
      events += self._body.oclAsType(Scope).allSubobjectsOfType(Cifx::Action)->
            oclAsType(Cifx::Action).map toEvent();

      if ( self.allSubobjectsOfType(Cifx::Automaton)->size() > 0 ) then {
         automata += self.allSubobjectsOfType(Cifx::Automaton)->oclAsType(Cifx::Automaton).map toAutomaton();
      } else {
         log("toModel: No automaton present.");
      } endif;

   } else {
      log("toModel: Body is not a scope.");
   } endif;
}

mapping Cifx::Variable::toVariable() : Ariadne::Variable {

   if ( self.container().oclIsTypeOf(Scope) and Prefix ) then {
      name := self.container().oclAsType(Scope).name + "_" + self.name;
   } else {
      log("Prefix: " + Prefix.repr() + ", Parent of the variable: " + self.container().metaClassName());
      name := self.name;
   } endif;

   log("toVariable: Transforming Variable " + name);

   if (self.staticType.oclIsTypeOf(Types::RealType)) then {
      type := Ariadne::StaticType::RealType;
   } else {
      log("toVariable: Variable not of RealType");
   } endif;

   if (self.initialValue.oclIsTypeOf(Expressions::LiteralExpression)) then {
      initialValue := self.initialValue.oclAsType(Expressions::LiteralExpression).map toLiteral();
   } endif;
}

mapping Action::toEvent() : Event {
   if ( self.container().oclIsTypeOf(Scope) and Prefix ) then {
      name := self.container().oclAsType(Scope).name + "_" + self.name;
   } else {
      log("Prefix: " + Prefix.repr() + ", Parent of the action: " + self.container().metaClassName());
      name := self.name;
   } endif;
   log("toEvents: Transforming Event " + name);
}

mapping Expressions::Expression::toExpression() : Ariadne::Expression
disjuncts Expressions::VariableReference::toExpVar,
      Expressions::ConstantReference::toExpConst,
      Expressions::UnaryExpression::toExpUnary,
      Expressions::BinaryExpression::toExpBinary,
      Expressions::LiteralExpression::toLiteral,
      Expressions::FunctionCallExpression::toFunction
{}
```

```
mapping Expressions::VariableReference::toExpVar() : Ariadne::VariableExpression {
    variable := self.variable.resolveone(Ariadne::Variable);
}

mapping Expressions::ConstantReference::toExpConst() : Ariadne::NamedConstantExpression {
    namedConstant := self.constant.resolveone(Ariadne::NamedConstant);
}

mapping Expressions::UnaryExpression::toExpUnary() : Ariadne::UnaryExpression {
    switch {
        case (self.operator = Expressions::UnaryOperator::Plus) operator := RealUnaryOperator::Plus;
        case (self.operator = Expressions::UnaryOperator::NumericalNegate) operator :=
            RealUnaryOperator::NumericalNegate;
    };

    child := self.child.map toExpression();
}

mapping Expressions::BinaryExpression::toExpBinary() : Ariadne::BinaryExpression {
    switch {
        case (self.operator = Expressions::BinaryOperator::Multiplication) operator := RealBinaryOperator::Multiplication;
        case (self.operator = Expressions::BinaryOperator::Power) operator := RealBinaryOperator::Power;
        case (self.operator = Expressions::BinaryOperator::RealDivision) operator := RealBinaryOperator::RealDivision;
        case (self.operator = Expressions::BinaryOperator::Subtraction) operator := RealBinaryOperator::Subtraction;
        case (self.operator = Expressions::BinaryOperator::Addition) operator := RealBinaryOperator::Addition;
    };

    leftChild := self.leftChild.map toExpression();
    rightChild := self.rightChild.map toExpression();
}

mapping Expressions::LiteralExpression::toLiteral() : Ariadne::ConstantExpression
disjuncts Expressions::RealNumber::toRealNumberConstant
{}

mapping Expressions::FunctionCallExpression::toFunction() : Ariadne::FunctionCallExpression {
    if (self.function.oclIsTypeOf(Expressions::StdLibFunctionReference)) then {
        function := self.function.oclAsType(StdLibFunctionReference).function;
        arguments += self.arguments->map toExpression();
    } else {
        log("toFunction: No function specified.");
    } endif;
}

mapping ConstantDeclaration::toConstant() : NamedConstant {
    name := self.name;
    log("toConstant: Transforming Constant " + name);

    if (self.value.oclIsTypeOf(Expressions::RealNumber)) then {
        value := self.value.oclAsType(Expressions::RealNumber).map toRealNumberConstant();
    } else {
        log("toConstant: Constant is not a RealNumber.");
    } endif;
}

mapping TopLevelDeclaration::fromTopLevelDeclaration() : NamedConstant
disjuncts ConstantDeclaration::toConstant
{}

mapping Expressions::RealNumber::toRealNumberConstant() : Ariadne::RealNumberConstant {
    value := self.value;
    log("toRealNumberConstant: Value " + value.repr());
}

mapping Automaton::toAutomaton() : Ariadne::Automaton {
    name := self.name;
    log("toAutomaton: Transforming Automaton " + name);

    --Note: The modes should be transformed first, because the transitions depend on them.
    --Otherwise, use a late resolve.
    transitions += self.edges.map toTransition();
    modes += self.locations.map toMode();
}

mapping Location::toMode() : Mode {
    name := self.name;
    log("toModes: Transforming Mode " + name);

    if (self.initial.oclIsTypeOf(Expressions::BoolLiteral)) then {
        initial := self.initial.oclAsType(Expressions::BoolLiteral).value;
    } endif;
```

```
    dynamics += self.flow->map toDynamic();
    algebraic  += self.invariant->map toAlgebraic();

    tcp += self.tcp->map toPredicate();
}

mapping Expressions::Expression::toDynamic() : Ariadne::DottedAssignment {
    --Only expressions allowed like "dot(x) = Expression"

    --The left child of the BinaryExpression should be an UnaryExpression with a
    --VariableExpression as child.
    if ( self.oclIsTypeOf(Expressions::BinaryExpression) ) then {
        if ( self.oclAsType(Expressions::BinaryExpression).operator = BinaryOperator::Equal ) then {
            if ( self.oclAsType(Expressions::BinaryExpression).leftChild.oclIsTypeOf(Expressions::UnaryExpression) ) then {
                if ( self.oclAsType(Expressions::BinaryExpression).leftChild.oclAsType(Expressions::UnaryExpression).
                    operator = Expressions::UnaryOperator::Derivative ) then {
                    if ( self.oclAsType(Expressions::BinaryExpression).leftChild.oclAsType(Expressions::UnaryExpression).
                        child.oclIsTypeOf(Expressions::VariableReference) ) then {

                        dottedVariable := self.oclAsType(Expressions::BinaryExpression).leftChild.oclAsType(Expressions::
        UnaryExpression).child.oclAsType(Expressions::VariableReference).variable.resolveone(Ariadne::Variable);

                        --Value, right child of the BinaryExpression.
                        value := self.oclAsType(Expressions::BinaryExpression).rightChild.map toExpression();

                    } else { log("toDynamics: Child of the UnaryOperator is not a VariableReference"); } endif;

                } else { log("toDynamics: UnaryOperator is not Derivative"); } endif;

            } else { log("toDynamics: leftChild is not an UnaryExpression"); } endif;
        } else { log("toDynamics: BinaryOperator is not Equal"); } endif;
    } else { log("toDynamics: No BinaryExpression"); } endif;
}

mapping Expressions::Expression::toAlgebraic() : Ariadne::Assignment {
    --Only expressions allowed like "x = Expression"
    --Left child of the BinaryExpression should be a VariableReference

    --When the argument of assert returns false, the message is displayed.
    --assert warning ( self.oclIsTypeOf(Expressions::UnaryExpression) )
    --with log ("Warning, wrong type of Expression.");

     if ( self.oclIsTypeOf(Expressions::BinaryExpression) ) then {
        if ( self.oclAsType(Expressions::BinaryExpression).operator = BinaryOperator::Equal ) then {
            if ( self.oclAsType(Expressions::BinaryExpression).leftChild.oclIsTypeOf(Expressions::VariableReference) ) then {

                variable  := self.oclAsType(Expressions::BinaryExpression).leftChild.oclAsType(Expressions::
                    VariableReference).variable.resolveone(Ariadne::Variable);

                --Value, right child of the BinaryExpression.
                value := self.oclAsType(Expressions::BinaryExpression).rightChild.map toExpression();

            } else { log("toAlgebraics: leftChild  is  not a VariableReference"); } endif;
        } else { log("toAlgebraics: BinaryOperator is not Equal"); } endif;
    } else { log("toAlgebraics: No BinExpression"); } endif;
}

mapping Edge::toTransition() : Transition {
var Results: OrderedSet(Ariadne::PrimedAssignment);

    if (self.action.oclIsTypeOf(ActionReference)) then {
        event := self.action.oclAsType(ActionReference).action.resolveone(Event);
    } endif;
    sourceMode := self.sourceLocation.late resolveoneIn(Location::toMode,Mode);
    targetMode := self.targetLocation.late resolveoneIn(Location::toMode,Mode);

    self.allSubobjectsOfType(Cifx::Update).oclAsType(Cifx::Update) ->
    forEach(Upd) {
        resets  += Upd.QueryFromUpdate();
    };

    self.allSubobjectsOfType(Cifx::Assignment).oclAsType(Cifx::Assignment) ->
    forEach(Upd) {
        resets  += Upd.QueryFromAssignment();
    };

    if ( self.guards->size() > 1 ) then {
        log("toTransition: More than one guard specified! Only the first  will  be transformed.");
    } endif;

    guard := self.guards->at(1).map toPredicate();
}
```

```
query Cifx::Update::QueryFromUpdate() : OrderedSet(Ariadne::PrimedAssignment) {
    var PrimedAssignments: OrderedSet(Ariadne::PrimedAssignment);

    if ( self.variables->size() <> self.expressions->size() ) then {
        log ("QueryFromUpdate: The number of variables does not correspond to the number of expressions.
        This is  not supported at the moment.");
    } else {
        self.expressions->forEach(Exp) {
            if ( Exp.oclIsTypeOf(Expressions::BinaryExpression) ) then {
                if ( Exp.oclAsType(Expressions::BinaryExpression).leftChild.oclIsTypeOf(Expressions::VariableReference) )
                then {
                    PrimedAssignments += object PrimedAssignment {
                        primedVariable := Exp.oclAsType(Expressions::BinaryExpression).leftChild.oclAsType(Expressions::
                        VariableReference).variable.resolveone(Ariadne::Variable);
                        value := Exp.oclAsType(Expressions::BinaryExpression).rightChild.map toExpression();
                    };
                } else {
                    log("fromUpdate: Left child is  not a VariableReference.");
                } endif;
            } else {
                log("fromUpdate: Expression is not a BinaryExpression.");
            }endif;
        }; --End forEach

    } endif;

    return PrimedAssignments;
}

query Cifx::Assignment::QueryFromAssignment() : OrderedSet(Ariadne::PrimedAssignment) {
    var PrimedAssignments: OrderedSet(Ariadne::PrimedAssignment);
    var Counter: Integer := 1;
    var Count: Integer;

    if ( self.addressable.oclIsTypeOf(AddressableVariable) ) then {
        log("Just one PrimedAssignment");

        PrimedAssignments += object PrimedAssignment {
            primedVariable := self.addressable.oclAsType(AddressableVariable).variable.resolveone(Ariadne::Variable);
            value := self.value.map toExpression();
        };

    } else {
        //So the addressable is either an AddressableTuple or an AddressableArray

        if ( self.addressable.oclIsTypeOf(AddressableTuple) and self.value.oclIsTypeOf(Expressions::TupleExpression))
        then {
            Count := self.addressable.oclAsType(AddressableTuple).elements->size();
            if ( Count = self.value.oclAsType(Expressions::TupleExpression).fields->size() ) then {

                while (Counter <= Count) {
                    PrimedAssignments += object PrimedAssignment {
                        primedVariable := self.addressable.oclAsType(AddressableTuple).elements->
                        at(Counter).oclAsType(AddressableVariable).variable.resolveone(Ariadne::Variable);
                        value := self.value.oclAsType(TupleExpression).fields->at(Counter).map toExpression();
                    };
                    Counter := Counter + 1
                }

            } else {
                log("fromAssignment: The number of variables does not correspond to the number of expressions.");
            } endif;
        } else {
            log("fromAssignment: AddressableArray not supported at the moment (or inconsistent data types).");
        } endif;

    } endif;

    return PrimedAssignments;
}

mapping Expressions::Expression::toPredicate() : Ariadne::Predicate
disjuncts Expressions::LiteralExpression::toConstantPredicate,
        Expressions::UnaryExpression::toUnaryPredicate,
        Expressions::BinaryExpression::toBinaryPredicate,
        Expressions::BinaryExpression::toComparisonPredicate
{}

mapping Expressions::LiteralExpression::toConstantPredicate() : Ariadne::ConstantPredicate {
    if ( self.oclIsTypeOf(BoolLiteral) ) then {
        value := self.oclAsType(BoolLiteral).value;
```

```
    } endif;
}

mapping Expressions::UnaryExpression::toUnaryPredicate() : Ariadne::UnaryPredicate {
    if ( self.operator = Expressions::UnaryOperator::BooleanNegate ) then {
        operator := BooleanUnaryOperator::BooleanNegate;

        child := self.child.map toPredicate();
    } endif;
}

mapping Expressions::BinaryExpression::toBinaryPredicate() : Ariadne::BinaryPredicate
when { (self.operator <> Expressions::BinaryOperator::GreaterEqual) and
(self.operator <> Expressions::BinaryOperator::LessEqual) }
{

    switch {
        case ( self.operator = Expressions::BinaryOperator::Conjunction ) operator := BooleanBinaryOperator::Conjunction;
        case ( self.operator = Expressions::BinaryOperator::Disjunction ) operator := BooleanBinaryOperator::Disjunction;
        case ( self.operator = Expressions::BinaryOperator::Implication ) operator := BooleanBinaryOperator::Implication;
    };

    leftChild := self.leftChild.map toPredicate();
    rightChild := self.rightChild.map toPredicate();

}

mapping Expressions::BinaryExpression::toComparisonPredicate() : Ariadne::ComparisonPredicate
when { (self.operator = Expressions::BinaryOperator::GreaterEqual) or
(self.operator = Expressions::BinaryOperator::LessEqual) }
{

    switch {
        case ( self.operator = Expressions::BinaryOperator::GreaterEqual ) operator :=
         ComparisonOperator::GreaterEqualThan;
        case ( self.operator = Expressions::BinaryOperator::LessEqual ) operator := ComparisonOperator::LessEqualThan;
    };

    leftChild := self.leftChild.map toExpression();
    rightChild := self.rightChild.map toExpression();
}
```