Ark, the Metamodelling Kernel for Domain Specific Modelling

Xiaoxi Dong
http://msdl.cs.mcgill.ca/people/xiaoxi

Supervisor: Professor Hans Vangheluwe

School of Computer Science
McGill University
Montreal, Quebec, Canada

A thesis submitted to the McGill University in partial
fulfilment of the requirements of the degree of
Master of Science in Software Engineering

Abstract

A model is an abstraction of the real system. To design complex systems, modelling is preferred to the traditional methods for its capability to analyse and simulate before implementation, and its tools for code generation which allows for defect-free code. The domain specific modelling and metamodels (the abstraction of models) provide the modellers domain specific syntax and environments. The meta-metamodel (the abstraction of metamodels) defines a unified description of various domain metamodels. Metamodelling architectures provide the guideline of organizing models and metamodels. So far, many metamodelling standards and tools have been developed.

However, two drawbacks have prevented us from having a well-defined metamodelling tool. The first is that current linear architectures fail to appropriately separate the views of different roles in meta-modelling. The second is the missing executability at the root of metamodelling since most existing meta-metamodels are designed to describe the structural information rather than the behavioural.

In this project, we used a two-dimensional metamodelling architecture with logical and physical classifications that separates the view of modellers and that of tool developers. We designed a general-purpose, self-describable, executable meta-metamodel ArkM3 which includes an action language and thereby enables executability. With this architecture and this meta-metamodel, we enabled a general-purpose, comprehensive, bootstrapped metamodelling tool. To demonstrate our design, we built Ark, the kernel of AToMPM (A Tool for Multi-Paradigm Metamodelling), an updated version of AToM^3 (A Tool for Multi-Formalism and MetaModelling). We also presented a case study that models a Readers/Writers System Petri Net model.

Un modèle est une abstraction dun système réel. Pour la conception de systèmes complexes, la modélisation est préférée aux méthodes traditionnelles, parce que la modélisation permet danalyser et de simuler avant la mise en œuvre. De plus, les outils de génération de code fournis par la modlisation aident à produire des codes sans défaut. La modélisation dans des domaines spécifiques et des méta-modèles (l'abstraction des modèles) fournissent des syntaxes et des environnements spécifiques aux modélisateurs. Le méta-métamodèle (l'abstraction de métamodèles) normalise la description des métamodèles. Les architectures de métamodélisation fournissent les directives à suivre afin dorganiser les modèles et les métamodèles. Jusqu'à ce jour, de nombreuses normes et doutils en métamodélisation ont été développés.

Cependant, deux inconvénients demeurent et préviennent la création dun outil de métamodélisation bien défini. Le premier est que les architectures actuelles sont linéaires. Ceci qui ne les permettent pas de différencier adéquatement l'aspect des différents rôles dans la métamodélisation. Le second est quelles nont pas de caractère exécutable à partir de la racine de la métamodélisation, puisque la plupart des méta-méamodèles sont conçus pour décrire des informations structurelles plutôt que comportementaux.

Dans ce projet, nous avons utilisé une architecture à deux dimensions avec la classification logique et physique, séparant ainsi le point de vue des modélisateurs et celui des développeurs d'outils. Nous avons conçu ArkM3, un méta-métamodèle universel, auto-descriptible et exécutable. Il inclut également un langage d'action, ce qui le rend exécutable. En combinant cette architecture et ce méta-métamodèle rend possible de mettre sur pied un outil universel damorçage de métamodélisation. Pour démontrer notre conception, nous avons construit le noyau de AToMPM (A Tool for Multi-Paradigm Metamodelling), une version mise à jour de AToM^3 (A Tool for Multi-Formalism and MetaModelling). Nous présentons également une étude de cas selon un système de Petri Net "Readers/Writers".

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Hans Vangheluwe, for introducing me to metamodelling and accepting me in the Modelling, Simulation and Design Lab in 2008. It was a great journey in the world of modelling. I had been discovering the enlightening view of metamodelling and had been learning and improving on every project. All would not have been possible without Hans' guidance in the course. Thanks, Hans!

I also would like to thanks my friends for their selfless support. I would not have enjoyed my three years at McGill University without their accompany and encouragement. Thanks, girls and guys!

Finally, I would not be writing these lines without my parents. Five years away from home, they have always been the solid rock on which I stood. Thank you, mom and dad!

Contents

List of Figures

List of Tables

1 Introduction

As the functional requirements advancing quickly, the size and the complexity of systems, either physical or software, grow significantly. The task of manual creation and maintenance of the source code becomes hardly feasible. On the one hand, the personal programming preference prevents the developers from co-operating with each other, and on the other hand, the difficulty in maintenance prevents the applications from upgrading. Therefore, the methodology and technology to rapidly produce high-quality, defect-free, and maintainable software became an important issue in software engineering [Kuh89]. In contrast with the classic code-based development technique, the use of models is more and more recommended.

A model is an abstraction of physical or software systems. By building a counterpart of a system, modelling makes design and analysis of complex systems possible. The designers can avoid the risk of implementing the systems before they have nicely designed and fully tested the models, so that they can produce the defect-free source code rapidly with automatic code-generation tools.

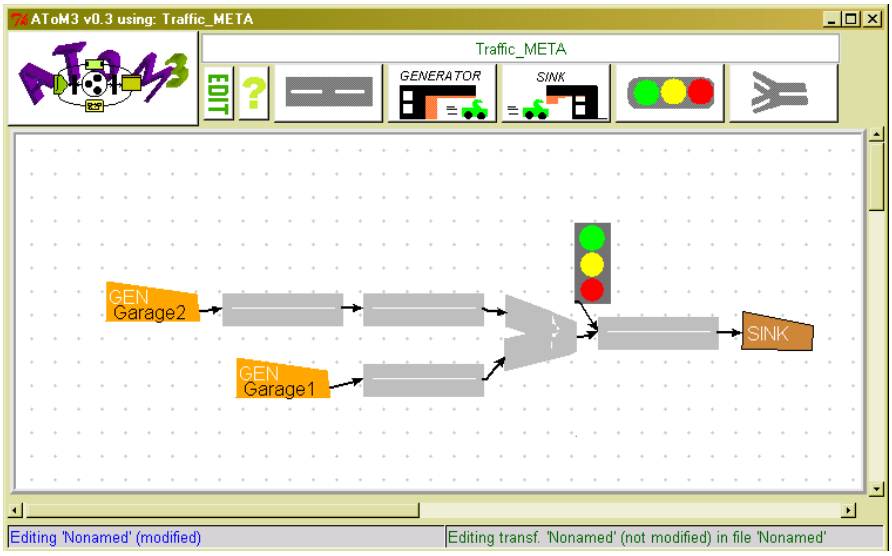


Figure 1.1: A Traffic Domain Specific Modelling Environment and a Traffic Model

Computer-Aided Software Engineering (CASE) tools dating back to the 1980s made the early attempts to automatic software development. Unified Modeling Language (UML) proposed in 1990s integrates a set of graphic notations that visually analyse and model system structure and behaviour[1]. Thereafter, the Domain Specific Modelling (DSM) and the metamodelling concept emerged around 2000. DSM focuses on the systems of a specific industry domain. The domain modellers define the syntax and semantics of

[1]General-purpose UML tools include both commercial tools such as Jude, Rose and free software such ArgoUML, Fujaba, BoUML, MagicDraw and a few web applications. They provide convenient analysis and documentation functions, as well as the code generation in different programming languages.

a domain (it is also called a domain specific language (DSL) or a formalism, usually graphical), whereby the users build systems. Figure 1.1 shows an example of a domain specific model built in AToM^3 (A Tool for Multi-Formalism and MetaModelling) [MSD09][2]. A traffic system is modelled in a Traffic DSM environment. This system has connected road segments, one traffic light, an joint, two traffic generator (source) and a traffic disposer (sink).

Generally speaking, the DSM supports an higher-level abstraction than that of general-purpose modelling languages. It requires less efforts and fewer low-level details to specify a given system.

As a complex system often has components and aspects whose structure and behaviour could be modelled and analysed via the most appropriate formalism, tools supporting Multi-Paradigm Modelling—that models one system with many formalisms—become necessary [dLV02].

Metamodelling is an essential enabler of the domain specific modelling and multi-paradigm modelling. A metamodel describes a DSL. It highlights the properties of the models, including the definition of the abstract syntax, the concrete syntax, and sometimes the semantics, that are sufficient to automatically generate a full domain- and problem-specific modelling, and possibly simulation, environment [MV02].

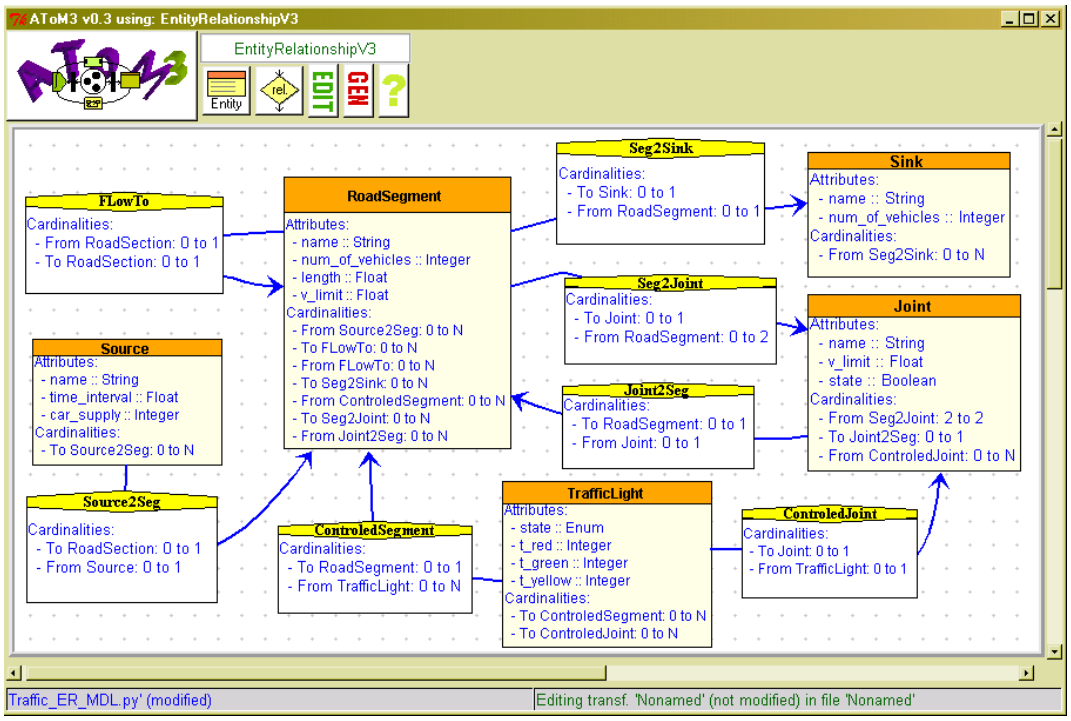


Figure 1.2: The Metamodel of the Traffic Formalism

A Traffic metamodel is as shown in Figure 1.2, which is modelled in a more general modelling formalism called Entity Relationship Model[3]. We identify entity types such as the road segment and the traffic light. Each of them has attributes and associates with other entity types, along with an icon to graphically represent this type in the generated environment. This metamodel is compiled and generates the Traffic metamodelliing environment in Figure 1.1[4].

[2] AToM^3 developed at MSDL in 2002 is one of many tools that support Domain Specific Modelling. After loading the formalisms into the environment, the users may draw models on the canvas. See Section 2.4

[3] Please refer to Section 2.1.3 for more information of ERM.

[4] We can find many other metamodelling tools emerged in the last two decades besides AToM^3. There are MetaEdit+ [SLTM91] (1990), DOME from Honeywell [SG08] (The official support of DoME provided by Honeywell ended in 2000 with version 5.3.), Xactium XMF, Metacase, Eclipse EMF [emf], VMTS [vmt] , Kermeta [Ker09] from INRIA. There are more to be mentioned in the following chapters of this thesis.

Metamodels can also have an abstraction, which is called the meta-metamodel. A meta-metamodel models the metamodels. It describes the syntax and semantics of the metamodels. It provides the common ground for model transformations. Many modelling standards and tools has defined the meta-metamodel as part of their framework to uniformly manage the metamodels. The common meta-metamodel used in AToM^3 is the Entity Relationship formalism as in Figure 1.3[5].

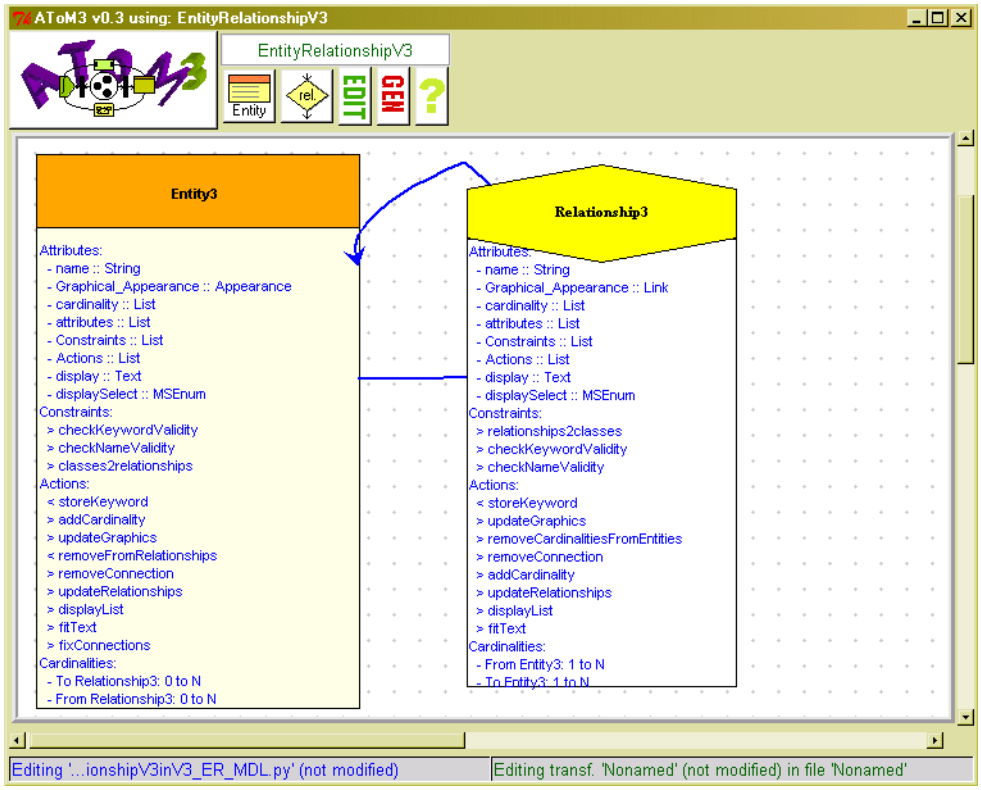


Figure 1.3: The Meta-metamodel Models the Traffic Metamodel

The relationship between the model and the metamodel, and the metamodel and the meta-metamodel is the "instance-of" relationship, or "typed-by" in the opposite. A metamodel is a model itself, while its metamodel is the meta-metamodel. The notion of meta- is relative [LV07]. In principle, one could continue the meta- hierarchy to infinity. Some meta-metamodel can be modelled by itself (indicating ERM in Figure 1.3 is modelled by ERM). This valuable feature of meta-circularity allows modelling tool and language compiler builders to bootstrap[6] their systems and therefore leads to platform- and programming language-independent models [VSB07].

Metamodelling architectures are the guidance to organize the models, metamodels and even meta-metamodels in accordance to the researchers' interpretation of model abstraction. There have been several metamodelling architectures proposed so far. The class-object architecture (so called golden braid) is the architecture used in the early research of SmallTalk [Coi87]. This nested metamodelling architecture is adopted in the Meta-Modelling Language (MML) [CEKI00] [Á01]. The Meta-Object Facility (MOF) Four-Layered Architecture [MOF02] is proposed by the OMG's (Object Management Group)

[5]There are also many meta-metamodel and metamodelling standards developed so far. Some of them are EIA/CDIF [Fla02] that enables sharing information between CASE tools, MOF [MOF06] that models UML, Eclipse EMF Ecore [BSME03] that customizes Java code generation, Kernel Meta-Metamodel (KM3) [JB06] that builds agile and precise models of the source, target and transformation for model transformation. There are more to be mentioned in the following chapters of this thesis.

[6]The term "bootstrap" is commonly used to describe the process by which a system starts up, which generally involves loading a small portion of the system in order to support the loading and initialization of its remainder.

Model-Driven Architecture (MDA) [mda03], a particular incarnation of MDE. MOF provides a layered, linear metamodelling architecture that follows the strict meta-modelling principle in which an element of a meta-layer is the instance of exactly one element at the upper meta-level. The layered architecture has been widely accepted by metamodelling tools. EMF [BSME03] trims the MOF architecture whereas VMTS [vmt] augments. AToM^3 allows arbitrary number of layers.

Although the above developments ease the task of building systems, there are pending issue. 1. It is not enough to merely model the structure of a system. Such models are not able to enforce more specific constraints on model elements over invariants, variable ranges and alike. Consequently, it results in an imprecise definition.

This problem was firstly attended in conceptual models for the information system using such as XML schema and SQL. It did not drawn much attention until the modelling being introduced. In order to have a well-formed model, modellers have made some attempts to include the constraints in the modelling process. One approach is to insert pieces of narrative languages and another to use formal language expressions. However, both approaches have their limitations. Narrative languages, or natural languages, are flexible by nature but the practice shows that using them always results in ambiguities. Formal language expressions are unambiguous, but they require the modellers to have solid mathematical background, which makes it a so very demanding task for general business and system modellers [OCL06]. Finally, Object Constraint Language(OCL) [OCL97] was introduced by IBM in the late 1990s to fill the gap. OCL is a specification language describing the rules and queries that apply to models and it is possible to adapt to various modelling tools and scenario.

2. Besides the constraints, researchers [DG06], [MFJ05], [LLMC05], [KPG07], [SGJ02], etc. found it important to capture the model behaviour. Adding executability to metamodelling is a natural evolution towards executable modelling. It enables the modellers to completely describe and simulate a system before implementation. It also allows for full code generation in desired programming languages. Programs have been used to describe actions. OCL is sometimes used to describe expressions for conditions. Executable formalisms such as Statecharts [Har87] and Petri Nets [CL08] can model the actions. UML Action Semantics (AS) [UML03] uses the sequence diagram to specify the model behaviour. Model transformation is used to analogue the change of model states. The above mentioned execution approaches have been proved to be useful in many applications. They have been adapted well in specific domains while have problems in others. These implementations are platform-dependent, specific in domains or hard to modify and update once they are implemented. Nevertheless, none of the above mentioned approaches include the executability in the meta-metamodel. The lack of a neutral executable action language prevents the operations of a model from being understood by other models. Including executability in the meta-metamodel is a trend in the metmaodelling researches. Some efforts have been done. There are standards and tools such as OMG's fUML [FUM08], Events and Properties (EP) [KPG08], Kermeta [MFJ05], Xion [MSFB05], and XMF [CSW08], etc.

3. In spite of many metamodelling architectures and their good acceptance in respective domains, there are some short-comings that limit their success.

Strict metamodelling principle is enforced to eliminate vagueness of the instance-of concept in the organization of the level hierarchy [dLG10], but there are situations that require more flexibility. To allow the flexibility while pertain the strictness, Atkinson and Khüne [AK08] introduced the potency to postpone the instantiation. This idea is further adapted by Juan de Lara and Esther Guerra's METADEPTH [dLG10].

However, the fact is that engineers often are forced to squeeze into two meta-modelling layers concepts that would naturally span several layers, resulting in more complex and cluttered models [dLG10]. As a solution, a dual ontological and linguistic instantiation is proposed, by precise UML [Á01] and Atkinson and Kühne [AK02], defining an element to be a linguistic instance and also an instance of some ontological domain concept.

In this project, our intention is to design an elegant, comprehensive, complete metamodelling architecture. Adapting the layered, strict architecture from MOF, and distinguishing the logical classification

and the physical classification, we proposed a two-dimensional metamodelling architecture that leads to general-purposed, platform-independent, comprehensive and bootstrapped metamodelling tools. The first dimension of this architecture describes the logical classification of the models and the second the internal physical representation. To increase the flexibility of the model while retain the strictness, instead of using potency and deep instantiation, we let the model import the required elements from the metamodel if it will not instantiate that element.

We also designed ArkM3, a self-describable meta-metamodel with executability. ArkM3's object meta-metamodel is inspired by the EMOF, the minimal set for metamodelling of MOF. To include advanced modelling concepts such as constraints and executability, we further modified the object model constructs to allow all the model elements to associate with Constraints and Actions. An action language meta-metamodel extends the object meta-metamodel. This action language is influenced by OCL [OCL06], Kermeta [DFV+09], and Modelica [CSW08] and some programming languages.

On the basis of the two-dimensional architecture and the meta-metamodel ArkM^3, we then implemented the kernel (Ark) of a metamodelling and transformation tool called AToMPM (A Tool for Multi-Paradigms Metamodelling) to validate our design. It comprises of modules to support creating, conformance checking of the models and the interpreter to interpret the action models. The hierarchical, attributed, directed, typed graph Himesis [Pro05] is adapted to represent the models in the physical dimension of a model. The mapping between the logical model and the physical HADT graphs is also specified.

Thesis Organization. Chapter 2 introduces the basic knowledge; Chapter 3 explains the design of AToMPM's two-dimentional architecture and meta-metamodel ArkM3; Chapter 4 presents the implementation of AToMPM kernel. Chapter 5 demonstrates an Petri Net model case study; and Chapter 6 draws the conclusion.

2 Related Work

2.1 Metamodelling

In this section, we give an overview of metamodelling, including the properties of metamodels, the metamodelling architecture, and the modelling of the constraint and the executability. We also discuss the importance of adopting a graph-based foundation for the metamodel representation. Finally, we will introduce the metamodelling tool AToM^3 (A Tool for Multi-Formalism and MetaModelling), which leads to our current work on AToMPM (A Tool for Multi-Paradigm Metamodelling).

2.1.1 Dissecting a Modelling Language

To explicitly model modelling languages, we will break down a modelling language into its basic constituents. This is illustrated in Figure 2.1, which is inspired by the description by Harel and Rumpe [HR00], taking common programming language concepts and putting them in a more general modelling context.

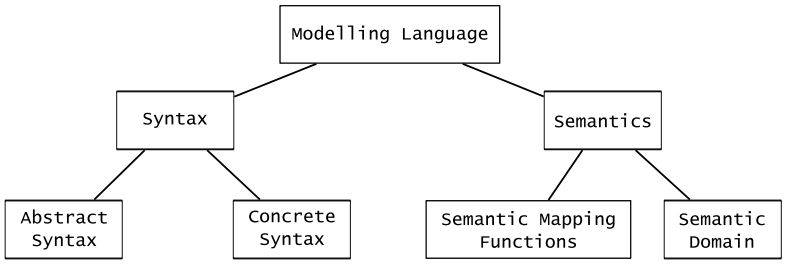


Figure 2.1: Modelling Language Breakdown, reproduced from [Pro05]

As explained in [Van08], the two main aspects of a model are its *syntax* (how it is represented) on the one hand and its *semantics* (what it means) on the other. The syntax of modelling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet*. These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (*i.e.,* belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language. For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the essence of the model. This is called the *abstract syntax*. A single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in *Abstract Syntax Trees* (ASTs). Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be *unique* and *precise* (to allow correct model exchange and code synthesis for example). Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own

right which needs to be properly modelled (and so on, recursively). In practice (in tools), the semantic mapping function maps abstract syntax onto abstract syntax.

Languages will explicitly be represented as shown in Figure 2.2. In the figure, insideness denotes the sub-set relationship. The dots represent models which are elements of the encompassing set(s). In the

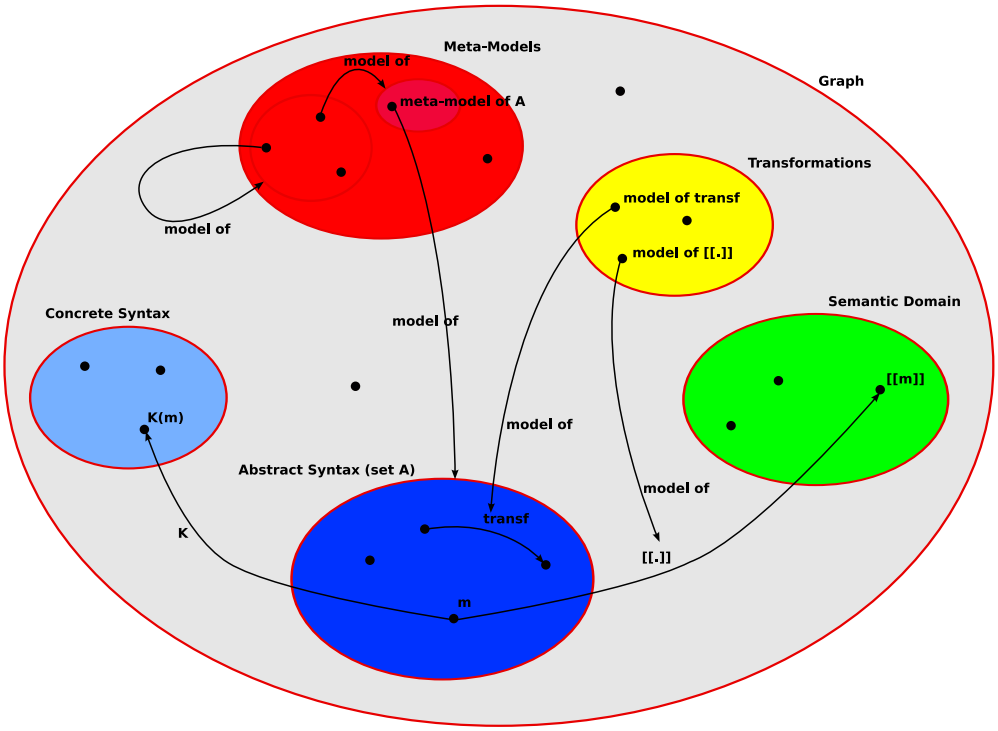


Figure 2.2: Modelling Languages as Sets, adopted from [VSB07]

bottom centre of Figure 2.2 is the abstract syntax set A.

Metamodelling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a finite model in an appropriate metamodelling language) of the abstract syntax set A of a modelling language. Often, metamodelling also covers a model of the concrete syntax. Semantics is however not covered. In the figure, the set A is described by means of the model metamodel of A. On the one hand, a metamodel can be used to *check* whether a general model (a graph) *belongs to* the set A. On the other hand, one could, at least in principle, use a metamodel to *generate* all elements of A. This explains why the term metamodel and grammar are often used inter-changeably.

A model m in the Abstract Syntax set (see Figure 2.2) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed. κ maps an abstract syntax graph onto a concrete syntax model.

Finally, a model m in the Abstract Syntax set (see Figure 2.2) needs a unique and precise meaning. This is achieved by providing a Semantic Domain and a semantic mapping function `[[.]]`.

The advantages of metamodelling are numerous. First, an explicit model of a modelling language can serve as documentation and as specification. Such a specification can be the basis for the analysis of properties of models in the language. From the metamodel, a modelling environment may be automatically generated. The flexibility of the approach is tremendous: new, possibly domain-specific languages can be designed by simply modifying parts of a metamodel. As this modification is explicitly applied to models, the relationship between different variants of a modelling language is apparent. Above all, with an appropriate metamodelling tool, modifying a metamodel and subsequently generating a possibly visual modelling tool is orders of magnitude faster than developing such a tool by hand. The tool

synthesis is repeatable and less error-prone than hand-crafting [VSB07].

2.1.2 Metamodelling Architecture

Among several metamodelling architectures, the so-called golden braid architecture and the MOF four-layered metamodelling architecture are representative. By either trimming or developing the two architectures, researchers have been proposing copious architectures to feature different industry domain. Researchers have been proposed multi-dimensional metamodelling architecture so as to achieve a clean separation of modellers' and tool developers' concerns in the term of describing model element[AK02].

Golden Braid

The golden braid architecture was inspired by the concept of bootstrapping discussed in [Hof99]. It was first adopted in an open ended system supporting ObjVlisp [Coi87], an early Object Oriented extension to SmallTalk [GR83].

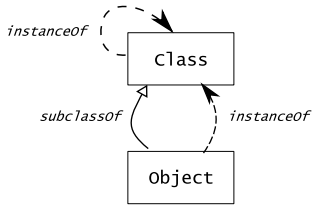


Figure 2.3: The Golden Braid Architecture

Considering the fact that metamodels, models and instances are all relative concepts based on the fundamental property of instantiation, the golden braid architecture describes the relationship between a Class and an Object. As shown in Figure 2.3,

- A Class can be instantiated to create an Object.
- A Class is a subclass of Object.
- As a consequence, a Class can also be instantiated from another Class: its "meta Class".

This architecture consists of two overlapping graphs, which are the *instance-of* graph (with *Class* as the root) and *subclass-of* graph (with *Object* as the root). By combining the two graphs we can create multiple meta-classes. There will be a distinct Class that elements from both graph in the meta-architecture are its instances. This class is effectively used to bootstrap the entire metamodel architecture. As shown in Figure 2.4, from *Class* many meta-classes can be defined and the graph will extend. *PolygonClass* class is an instance of *Class* and at the same time is a subclass of *Class*. *Hexagon*, *Polygon* and *Square* are instances of *PolygonClass*. *Hexagon* and *Square* inherit from *Polygon*, which inherits from *Object*.

The golden braid architecture does not have limit on the number of times that the instantiation relation is used, therefore offers great flexibility. This architecture provides the same structure to both the kernel and the models, and as a result, it does not have clear boundary between the tool and the models. The Golden braid architecture gives developers the access to the kernel and enables the modification of critic process such as instantiate and inheritance. This feature is, on the one hand, provides convenience, but on the other hand, increases the possibilities of users abusing the modelling environment.

Besides being used in SmallTalk [GR83] where the architecture was firstly proposed, the golden braid architecture is also adopted in XMF [CSW08] for its simplicity.

MOF Four-Layered Metamodel Architecture

Originally devised for the ANSIs Information Resource Dictionary System [Ins89], heavily influenced by EIA/CDIF [Fla02], the Meta Object Facility (MOF) [MOF06] is an adopted OMG (the Object Management Group) standard in the Model-Driven Architecture (MDA) [mda03]. It provides a model

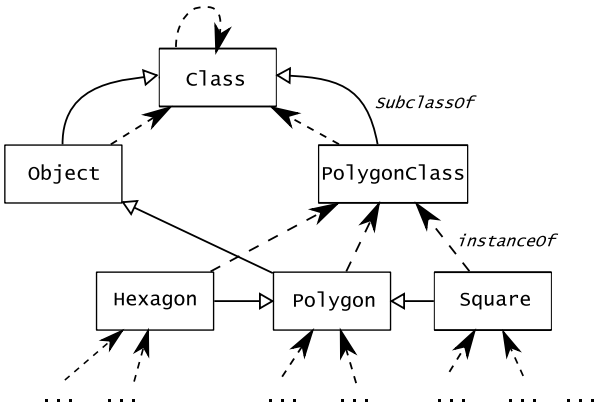


Figure 2.4: Example of Extending the Golden Braid Architecture

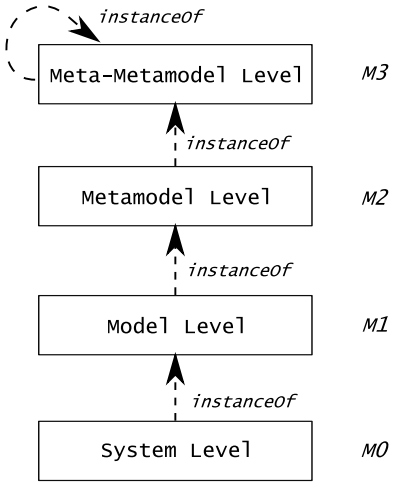


Figure 2.5: MOF Four-Layered Metamodel Architecture

management architecture and a set of model services to enable the development and interoperability of models and model driven systems. The metamodel architecture adopted in MOF is often called the MOF Four-Layered Metamodel Architecture in OMG specifications. It separates the models into layers in accordance to the "instance-of" relationship.

- M0 User-Data Level: contains the data of the application. For example, a computer game that is object oriented or a finite state machine at run time. This level is also called the user-object level.
- M1 Model level: contains the abstraction of the application. For example, the class definitions of a system, or Statecharts describes the behaviour of a finite state machine. This is the level at which application modelling takes place.
- M2 Metamodel level: contains the metamodel that captures the modelling language. For example, UML elements such as Class, Attribute, and Operation; definition of Statecharts State and Transition. This is the level at which metamodelling tools operate.
- M3 Meta-metamodel level: The meta-metamodel that describes the properties of all meta-models can exhibit.

The root in this architecture is the meta-metamodel. It defines the smallest set of concepts required to define meta-models, including itself. For example, as in the example in Section 1, a traffic system simulation is M0, the traffic model in Figure 1.1 is M1, the ERM model describing traffic system in Figure 1.2 is M2, and the ERM model that describes ERM itself in Figure 1.3 is M3.

Eclipse Modelling Framework (EMF) [BSME03] is a notable implementation of MOF. EMF, a "MDA on training wheels", sits in between programming and metamodelling. Instead of by general meta-metamodels, EMF describes models in Java, XML or UML. Comparing to MOF, EMF has M0, M1 and M2 levels, but its M3 level is yet of "instance-of" relationship with M2, as M3 should have been a finite representation of Java. EMF is truly integrated with and tuned for efficient programming since it is designed especially for the Java developing environment Eclipse. EMF relates modelling concepts directly to their implementations, thereby brings to Eclipse and Java developers the benefits of modelling with a low cost of entry.

Kermeta [MFJ05] also adopts the MOF architecture to make use of its standard and fully developed tools.

VMTS [LLMC05] uses an N-layered architecture. This architecture adds to the MOF architecture, alongside the model, metamodel and meta-metamodel layers, one layer for the read-only metamodel which specifies the metamodelling language(M^4) and the other layer the internal structure (a labelled directed graph) for model storage.

AToM^3 [MSD09] has arbitrary number of metamodelling layers. Although the Entity Relationship formalism is often used as the meta-metamodel, there is not one formalism that is defined as the top level metamodel. The user are free to include as many "instance-of" relationship as it is necessary. A newer version of AToM^3 (METADEPTH) [dLG10] considers more than two layers at a time by using the notion of *potency* [AK08].

MOF Four-Layered Metamodelling Architecture and AToM^3 architecture are somehow similar especially after it claimed in MOF 2.0 Specification that, "Note that key modelling concepts are Classifier and Instance or Class and Object, and the ability to navigate from an instance to its meta-object. This fundamental concept can be used to handle any number of layers. MOF 2.0 with its reflection model can be used with as few as two levels and as many levels as users define."

MOF Four-Layered Metamodelling Architecture and the golden braid architecture are different in the strictness [1]. MOF enforces on the *instance-of* relation between layers. The models and instances are in the different layers. It helps, in the industry, to separate responsibilities of modellers and metamodellers. On the contrary, the golden braid architecture is flexible that the metamodels and models are not strictly separated. The tools that build on this need extra mechanisms to prevent the users from accessing core models of the tools.

Strict Metamodelling

The strict metamodelling principle is introduced to eliminate vagueness of the *instance-of* concept in the organization of the level hierarchy [AK01].

Strict metamodelling distinguishes the relation of models at different metamodelling levels and that of models at the same level. The relations of individual model elements are illustrated schematically in Figure 2.6. All the elements in level N are strictly the instances of the elements in level N+1, that is the models at some level must be typed by the model in the higher level, while all the elements in the lower level must be an instance of some elements in the higher level. Any relation other than the *instance-of* relation between two elements implies that it is the same level. As a result of applying this principle, the metamodelling levels are formed purely by *instance-of* relation. Every modelling element can be assigned its proper location. Without strict metamodelling, the multilevel hierarchy would collapse into a single level [AK02].

[1] The concept of strictness is explained in Section 2.1.2

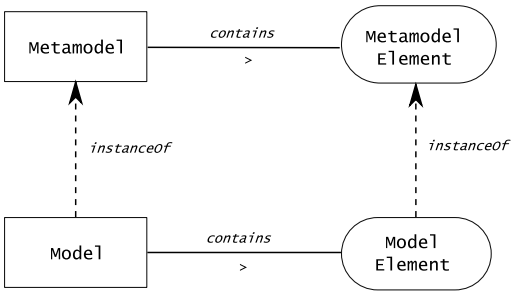


Figure 2.6: Strict Metamodelling

The strict metamodelling also implies that there is no top level of the metamodelling hierarchy when the top level metamodel can be described as an instance of itself. Therefore, a closed architecture can be constructed based on this principle.

Since strictness appeared to provide a foundation upon which a sound metamodelling hierarchy could be established, and offered a discipline for the development of metamodels, adherence to strictness has been recommended since UML 1.4. It is also worth extending to future metamodelling architectures.

It is possible for a model to require metamodels on more than one metamodelling layers. Because of the strict metamodelling, We can not simply instantiate those metamodels in different layers. *Deep instantiation* and *potency* are introduced by Atkinson and Kühne [AK01] [AK08] to solve this dilemma. Deep instantiation allows information to be carried over more than one instantiations. The potency is a nature number. It indicates how many times they can be instantiate. We can assign a potency to model elements. Each time when the model is instantiated and we go down a meta-level, the potency decreases by 1. The element is not assigned a value (i.e. become a plain instance) until its potency equals to 0. Deep instantiation is adopted by metamodelling tool METADEPTH [dLG10].

This approach ensures that the relationship between layers is purely "instance-of". Nevertheless, to correctly understand a model, one needs to know all the metamodels—they can span over several metamodelling layers. This adds to the complexity for understanding the models. The model is hard to exchange between tools as it requires transporting all the metamodels.

Multi-dimensional Metamodelling Architecture

The metamodelling architectures introduced in the previous section assume that metamodelling is a linear process, that is, a model can only have one metamodel. More recently, Atkinson and Khüne proposed [AK02] the multi-dimensional metamodelling architecture. They analyse the different responsibility in the modelling process, and suggest that the ambiguity of metamodelling is due to tool designers' failure to properly recognize and accommodate two fundamentally distinct forms of classification—the logical classification and the physical classification.

The logical classification is from the viewpoint of a modeller who takes care of composition and association in a particular domain. The physical classification corresponds to how the model is represented in the modelling tool. It is useful from the viewpoint of a tool builder who focuses on improving the performance of the modelling tool such as accelerating the searching process, reducing memory cost, etc. For example, we can consider the VMTS labelled graph layer mentioned in Section 2.1.2 as a physical classification. It is suggested that explicitly identify the *instance-of* relation as being either physical or logical is the first step to a coherent modelling framework. Figure 2.7 illustrates the two dimensions.

Along the logical axis, as we know from our daily experience, *Shrek* is a *Film*. We also know *Film* is a certain type of commercial product. *Film* is an instance of *ProductType*, and *Shrek* is an instance of *Film*. The hierarchy is based on the domain logic, but does not necessarily have any relation with how we maintain this structure. Meanwhile, along the horizontal axis, the entities in the logic view are considered instances of the *Structural Element* in *P1*. It is the projection from logical view to physical,

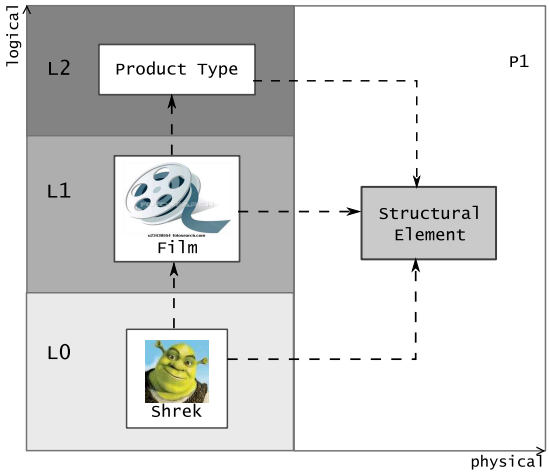


Figure 2.7: Example of A Two-Dimensional Modelling Architecture, reproduced from [AK02]

implementation view. The software developer, different from the modellers, is not interested in the name or the type of the domain entities. In the software, they are objects and connections with properties and behaviours. Using such abstraction helps tool developers to focus on optimizing their implementation.

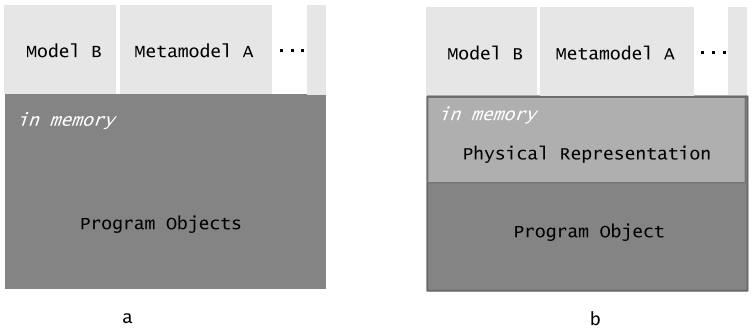


Figure 2.8: Add a Physical Representation between Model and Program Objects

Traditionally, as shown in Figure 2.8-a, the logical models (including metamodels and meta-metamodels) can be represented directly by objects of an object-oriented programming language. In this case, the meta-metamodel designers have to define the mappings between models and different programming languages constructs respectively. This results in poor interoperability. Distinguishing the physical dimension adds an intermediate layer of neutral representation between the models and the code as Figure 2.8-b. All the logical models and metamodels are represented by this physical representation in the metamodelling tool. Although the tool maps the physical representation to various formats, the modellers only need to define mappings from models to the physical representation.

A Multi-dimensional metamodelling framework conceptually adds a new aspect to metamodelling. It improves the model portability as models in the logic classification are tool-independent for free. It is an important step towards a conceptually clean and maintainable multi-paradigm modelling architecture. The research into multi-dimensional architectures is not complete yet. Besides dividing the concerns into logical and physical, it is also possible to add more dimensions or modify the meaning of the dimension[AGK09].

2.1.3 MOF, Ecore and Many Other Meta-Metamodels

On the third level of the MOF metamodelling architecture, the meta-metamodel level is located. The meta-metamodel is the core model around which a metamodelling tool is built. It is the general, neutral definition in which all the metamodels we can define. When a meta-metamodel can be typed by itself, it enables the self-described, bootstrapped metamodelling tools. So far, both programming languages and general, high-level meta-metamodels have been used for this purpose.

Programming languages are used for metamodelling in some tools. It is not as high level of abstraction as that of a meta-metamodel, since the meta-metamodel should actually model a programming language. Nevertheless, tool developers use programming language in favour of the performance. Popular programming languages have efficient compilers and virtual machines. Modellers can also get assistance from their IDE such as browsing, debugging, versioning, testing, refactorying, etc. Using programming languages also lower the barrier for the modellers who have background in programming. Moreover, describing models using a programming language naturally leads to executable models. A reflective language makes it possible to query the representation of the language itself and to modify it [DG06]. MOOSE [DG06] is a meta-described re-engineering environment uses SmallTalk as the executable meta-metamodel. EMF adopts Java as it provides a metamodelling environment based on Java development environment.

Unfortunately, using programming languages has a few disadvantages. Firstly, programming languages, including object-oriented programming languages, do not directly support associations, derived entities and opposite properties. Secondly, the modelling has to depend on the features of a particular programming language, and it results in non-portable tools. Therefore, a general, platform- and language-independent meta-metamodel is preferable to transport and bootstrap a metamodelling environment. Meta-metamodels use neutral descriptions, so that the models can transform between formalisms and platforms. Meta-metamodel's explicit, sometimes graphical, notations make it easier for general modellers to start with.

Among all these different meta-metamodels, we are going to introduce Entity Relationship Model, MOF, and Ecore. Among them, Ecore and MOF have been well accepted by the industry, and Entity Relationship Model formalism describes the fundamental composition of the object-oriented models. These meta-metamodels have already had supporting tools. There are also other meta-metamodels derived from these three meta-metamodels. We will introduce some of them later in Section 2.2.

Entity Relationship Model

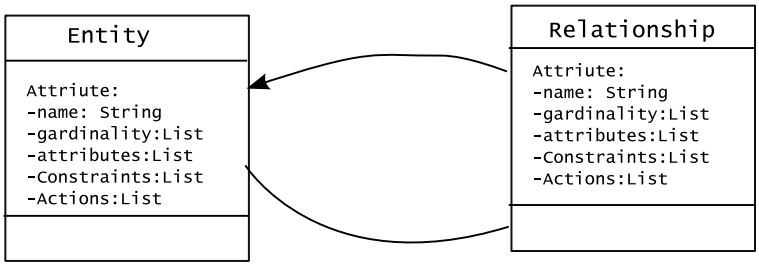


Figure 2.9: The Entity Relationship Metamodel

Entity Relationship Model (ER) was conceived by Peter Chen [sC76] as a database modelling method based on the set theory and relation theory, but the idea had also been used to produce conceptual models.

The entity relationship model adopts the natural view that the real world consists of entities and relationships. It is a simple formalism that, as shown in Figure 2.9, only has two types of elements: *entities* and *relations* that links the entities. An entity is a "thing" which can be distinctly identified, such as a car or a specific person. A relationship captures the relation between entities. Both entities

and relationships have properties. A relationship has mapping multiplicity. The concrete syntax of ER is usually expressed in a diagram.

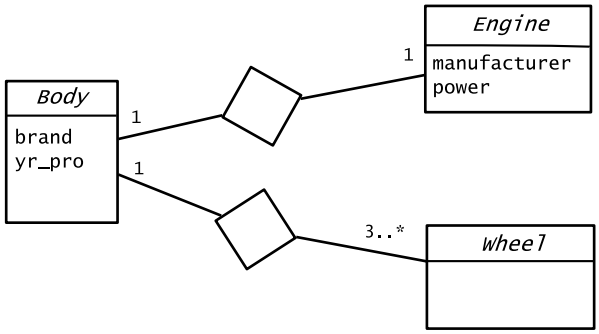


Figure 2.10: An Entity Relationship Model Example

Besides describing data, ERM can serve as a basic model for object-oriented design. For example, we can describe cars as shown in Figure 2.10, where a car consists of a body, wheels and an engine. The body has two attributes, *brand* and year of production *yr_pro*. The engine has *manufacturer* and *power*. One car can have many *(3..)* wheels but only one engine *(1)*.

Simple and general, ERM can describe any object-oriented models and even the syntax of the ERM formalism itself. This makes ER a good candidate for a stand-alone, bootstrapped meta-metamodel. ER is not expressive enough for very complex systems. The size of ER model would grow rapidly and the speed of traversing and analysing very large models could be compromised. A number of extension were introduced, such as EER (Extended ER) that introduced the generalization/specification relationship [TYF86][2] and HERM (Higher-Order Entity-Relationship Model) [Tha00] that adopts the cluster of types and higher-order relationship between entity types and clusters.

EMOF: Essential Meta-Object Facility

The MOF is a platform-independent mechanism that can model other MDA specifications as well as itself. MOF 1.4 [MOF02] was introduced in 2002 and the current version is MOF 2.0 [MOF06]. MOF builds on a subset of UML that provides the concepts and the graphical notations to the MOF model. It consists of two main packages, the Essential MOF (EMOF) and the Complete MOF (CMOF), along with the support for identifiers, additional primitive types, reflection and simple extensibility through name-value pairs (they are defined in separate packages). EMOF is a subset of MOF. The primary goal of EMOF is to allow simple metamodels to be defined using simple concepts. CMOF is built from EMOF and the Core::Constructs of UML which defines basic metamodelling capabilities. EMOF could be extended for more sophisticated metamodelling using CMOF. We, in this project, are more interested in EMOF, as what we want is a minimal, yet enough set of definitions to model object-oriented languages.

EMOF merges the Core::Basic package from UML and includes additional language capabilities by merging corresponding packages. EMOF merges the Reflection, Identifiers, and Extension capability packages to provide support for discovering, manipulating, identifying, and extending models. EMOF also provides a straightforward framework for mapping MOF models to implementations such as JMI and XMI.

It is noteworthy that EMOF is a separate model that **merges** the above packages instead of **extends** them. The goal is to ensure that EMOF has full support of itself. EMOF is completely specified in itself after applying the package merge. Therefore, it can be used to bootstrap metamodelling tools without requiring an implementation of CMOF and package merge semantics. Figure 2.12 shows the class diagram of EMOF after merging Core::Basic and the MOF capabilities.

An *Element* is a constituent of the model. *Element* is abstract and does not have superclass. It is the

[2]ER modelling aims at natural application representation which means no artificial or abstract type should be used.

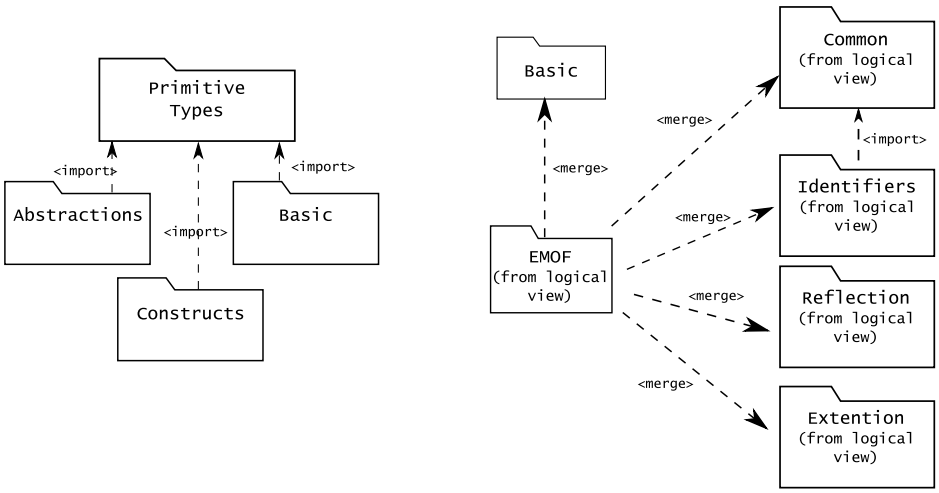


Figure 2.11: EMOF Merges Packages

common superclass for all the classes in the UML infrastructure library. The *Type* defines an abstract class that deals with naming and typing of elements. Names are required for all *Types*. A *Class* is a subclass of *Type* that has objects as instances. *Class* have properties and operations. When a class is abstract it cannot have any direct instances. A *Class* has attributes. The attributes are represented by instances of *Property*. A *Property* is a *TypedElement* that represents the attribute of the class. A property has a type and a multiplicity. When a property is paired with an opposite, they represent two mutually constrained attributes. "Opposite" is equivalent to a by-directional association. *Class* has operations. An *Operation* invokes on any object that is directly or indirectly an instance of the class. It is a typed element and a multiplicity element. *Operation* has ordered parameters. It can be associated with a set of types that represent possible exceptions this operation may raise during the invocation.

EMOF has characteristics that are essential to an object-oriented design: inheritance and encapsulation. As for the inheritance, *Class* instances participate in inheritance hierarchies. A class can have *superClass*. Any direct instance of a concrete (i.e., non-abstract) class is also an indirect instance of its class's super classes. An object permits the invocation of operations defined in its class and its class's super classes. Any direct instance of a concrete (i.e. non-abstract) class is also an indirect instance of its super classes. Moreover, EMOF allows for multiple inheritance. As for the encapsulation, a class cannot access private features of another class, or protected features of another class that is not its super type.

A *Package* is a container for types and other packages. Packages provide a way of grouping types and packages, which can be useful for understanding and managing models.

The *DataType* define data types. EMOF supports *PrimitiveType* including Boolean, Integer, and String, as well as the user defined *Enumeration*[3].

Although MOF is a wide spread metamodelling standard, but however, despite its self-describable feature, EMOF is less used as the meta-metamodel. It is because many consider that EMOF is the minimal set of model elements and does not have enough expressiveness.

Ecore

Ecore is the meta-metamodel of the Eclipse Modelling Framework (EMF) [BSME03]. EMF is an open source and open standard modelling framework designed towards Eclipse developed in around 2002. As introduced in Section 2.1.2, EMF is a powerful MOF-like framework that enables code generation, model and metadata management. It unifies Java, XML, and UML technologies, so that they can be

[3]The definitions of Boolean, Integer, and String are consistent with the XML schema [MOF06] in *http://www.w3.org/TR/xmlschema-2/*.

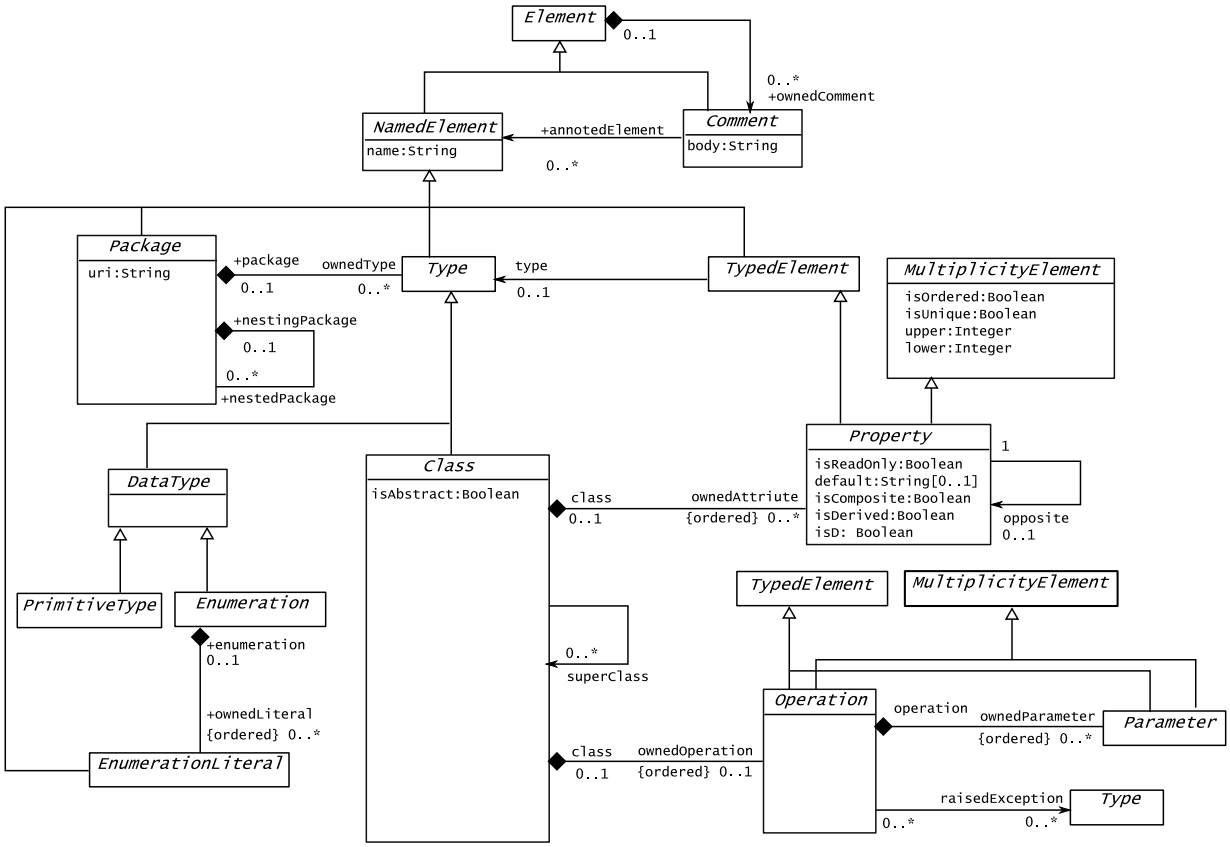


Figure 2.12: EMOF Class Diagram

used together to build better integrated software tools [BSME03].

Ecore is the meta-metamodel in EMF. Ecore has its root in MOF and UML, and is designed to map cleanly to Java implementations. Ecore is more or less aligned to OMG's EMOF. An EMF model is essentially the Class Diagram subset of UML [BSME03]. Ecore acts as its own metamodel. The users can treat Ecore like any other EMF models and benefit from the EMF generator in creating and maintaining Ecore implementation. The Ecore abstract syntax is as defined in Figure 2.13. *EClass* models classes. A class is identified by the name and contains attributes and references. To support the inheritance, a class can refer to a number of other classes as its *eSuperTypes*. *EAttribute* models attributes, the components of an object's data. An attribute is identified by name and has a type. *EDataType* models the types of attributes. Note that the primitive types and object data types are defined, instead of in EMF, in Java. *EReference* models associations between classes. Containment is a stronger type of association and the reference specifies whether to enforce this semantics by *containment*.

There are three sets of concrete syntax to describe Ecore models: Java, XML and UML. An EMF model can be created in any one of the three and the EMF tool can generate code in the others. Regardless of which one is used to define it, an EMF model is the common high-level representation that "glues" them all together [BSME03]. Closely related to the Java environment, EMF has good performance on code generation and model simulation. However, although EMF designers believe that the EMF represents the right level of abstraction, Ecore is not a platform-independent meta-metamodel.

ER, MOF/EMOF and Ecore have been used widely as the basis of other meta-metamodels, and as the composed object meta-metamodel in many other metamodelling architectures and tools. However, there has been discussions on the limitations of these meta-metamodels [MFJ05]. The discussion mainly focus on the fact that most meta-metamodels can not describe actions (the terms behaviour and executability

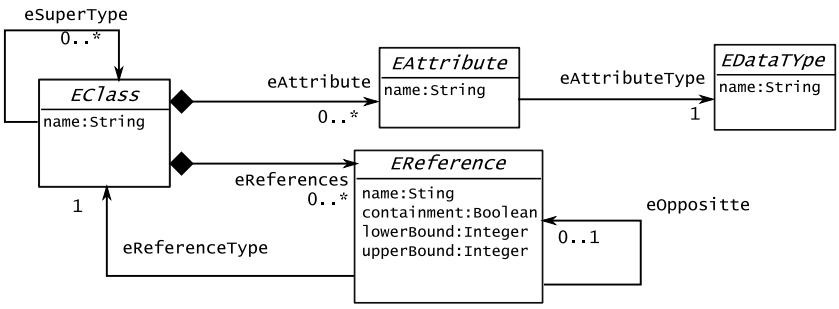


Figure 2.13: Ecore: the EMF Meta-Metamodel

are also used). In the next section, we will introduce the current development in model executability.

2.1.4 Constraint Language

Object Constraint Language(OCL) [OCL97] is a specification language describing the rules and queries that apply to models. The OCL Version 2.0 specification shares a common core with the UML Infrastructure and the MOF Core specifications that were developed in parallel. OCL is completely programming language-independent, which makes it possible to adapt to various modelling tools and scenarios. It serves as a complement to UML, MOF and QVT (Query/View/Transformation, a model transformation standard [QVT08]) to describe the constraints of models. It is easily used for specifying invariants on classes and types for MOF classifiers. It is also adopted to describe pre- and post- conditions on operations and specify the guards. It refers to model classifiers using a navigator which makes OCL a navigation language for graph-based models. It is reusable and is used as sub-expressions for other OCL expressions. Its primitives make it possible to describe constraints of query operations.

Being widely used and tested in describing operational behaviour, OCL types and operations have been good references for a new meta-metamodel. OCL provides rich data types. It has the primitive data types such as Integer, Read, Boolean, String and unlimited Integer, and the operations of these types. It also supports collection-related data types that are Set, Ordered Set, Bag and Sequence and their operations such as adding, deleting, union and intersection. To represent a type that conforms to all types, OCL provides VoidType. To combine different types into a single aggregate type, OCL provides TupleType.

OCL has provides a set of predefined iterator expressions which have a collection as their source. The following expressions are available for all collection types.

operation	*description*	*syntax*
exists	Results is true if body[4] evaluates to be true for at least one element in the source collection	source->exists(iterators | body)
forAll	Results is true if the body expression evaluates to be true for each element in the source collection	source->forAll(iterators | body)
isUnique	Results is true if body evaluates to be a different value for each element in the source collection	source->isUnique (iterators | body)
forAll	Results is true if the body expression evaluates to be true for each element in the source collection;	source->forAll(iterators | body)
any	Returns any element in the source collection for which body evaluates to true.	source->any(iterator | body)
collect	The Collection of elements that results from applying body to every member of the source set.	source->collect (iterators | body)
one	Results in true if there is exactly one element in the source collection for which body is true.	source->one(iterator | body)

The following expressions are defined individually for Set, Bag and Sequence

operation	*description*	*syntax*
select	The subset of set for which body is true.	source->select(iterator| body)
reject	Return the subset of the source set for which body is false.	source->reject(iterator | body)
sortedBy	Results is an OrderedSet containing all elements of the source collection.	source->sortedBy(iterator | body)
collectNested	Return a bag of elements that results from applying body to every member of the source bag.	source->collect(iterators | body)

OCL's close relation to the UML specifications provides powerful support of copious tools and a significant user base. Its navigation mechanisms are efficient, platform independent, and allow expression of complex queries. The majority of contemporary model management languages and tools use a subset of OCL for navigation and expressing constraints. However, OCL has its limitations as discusses in [KPG07] [KPP06],etc. OCL does not support statement sequencing. This must be encoded using nested and quantified expressions. In consequence, complex statements which are difficult to understand and maintain. Although OCL expressions can refer to model classifiers, modellers cannot invoke processes or activate non-query operations within OCL, which means evaluating these expressions will not change the states of the model. OCL is insufficient to describe behavioural models.

2.2 Model Executability

In this section, we will use the following small example to illustrate the use of various approaches. This example consists of query, constraint and model state modification, i.e, all the important features we want to examine in a executable model.

```
Consider a model object A which is a Facebook account having a Boolean
attribute P indicating the popularity. When the number of friends N of A
is lower than 300, the account popularity A.P is set to False otherwise
it is not.
```

Traditional Approaches

One of the approaches is to insert pieces of programs into the models. Many modelling tools use this strategy. The AToM^3 Transformation formalism integrates Python programs and the modellers need to understand how the model objects are stored in memory in order to access model elements. EMF either generates Java source code or dynamically loads generated Java classes (as a JIT compiler does in executing a Java program).
The Facebook example is as such if we expressed it in Python program,

```
if  type(A) == 'Account':
    if A.N <300:
        A.P = False
    else:
        A.P = True
```

The advantages of using code lie in that the speed of compiled binaries is much higher than interpreted code; less memory is required; the models are type-safe as the code can be checked by the compilers. Nevertheless, the disadvantages of this approach can not be ignored. It is platform- and language-dependent. Programs are hard for general system and business user to understand. The languages provide both too much and too few features. It is not an easy way to simultaneously restrict and extend

such existing languages [MFJ05]. It is also hard to maintain when requirements change and exchange models between tools.

OCL is often used to describe expressions for conditions. However, OCL only queries the status rather than changes them. The Facebook example can not be described using OCL because OCL can only check if A.N is larger than 300 but can not assign new value to A.P.

Although OCL can not express the action alone, it has been adapted by many recent implementations in executable metalanguage. For example, Xion [MSFB05] extends OCL with imperative components to describe actions in the web modelling context. The XMF [CSW08] mentioned in the next section also modifies OCL by adding executability.

Executable formalisms have been developed to describe system behaviour. These formalisms are designed to specify the actions for specific types of systems. They can be used by other formalisms. For example, a Statecharts [Har87] model can be a class attribute. The Facebook example can be described by Statecharts as in Figure 2.14. The green circle is the initial state and the transition is fired when the condition $A.N > 300$ is true.

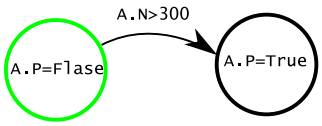


Figure 2.14: Change Model State using Statecharts

We can also choose other formalism such as the Discrete Event System (DEVS) [ZKP00] that models the discrete system as a state machine; the Adaptive Object Models [YBJ01] that describes business rules and view via the work flow diagram; the Event-Condition-Action Rule-based action specification [GKD01] that is often used in the domain such as web and web semantics.

These formalisms are at a higher level of abstraction than that of imperative programming languages, but this approach does not grant meta-metamodel and metamodelling tools the power of describing general behaviour and executing the action in a language-independent manner.

The Action Semantics (AS) for the UML aims at integrating a precise, implementation-independent action specification into the UML (thereby UML becomes xUML, or executable UML, or UML with Action Semantics). UML Action Semantics [UML03](part of the UML 1.5) provides a complete set of actions at a high level of abstraction. It is a platform-independent language for executing models. It is a set of models represented by diagrams. It defines fine-grained general-purpose actions [MFJ05]. Figure 2.15 illustrates how to use activity diagrams to describe the example. Other UML diagrams may also be used.

Action Semantics also has some drawbacks. It is a complex language which includes constructs that are not relevant to metamodelling such as concurrent actions used in multi-threaded and distributed systems. Moreover, UML Activity Diagrams become large very rapidly and it is intractable to draw and hard to comprehend thereafter.

Model transformation is used to support model executability. The modellers can either transform the model continuously as if they are executing (the source and target formalism are identical) or transforms the model to an intermediate formalism that is executable.

Rule-based model transformation allows the modellers to specify the transformation as a set of transformation rules that is, again, at a higher level of abstract than that of a imperative programming language. It comprises a set of transformation rules each with a certain priority. The model structure is changed by these rules and so are the properties of the model elements.

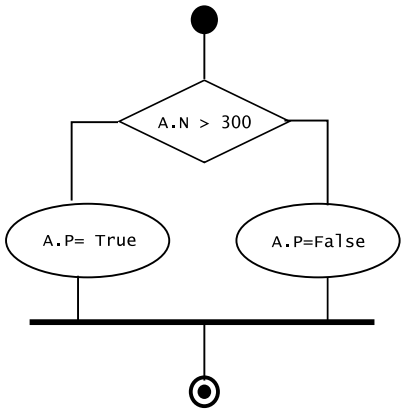


Figure 2.15: UML Action Semantics

A transformation rule can be expressed by either rewriting logic or graph transformation. Graph transformations have originally evolved in reaction to shortcomings in the expressiveness of classical approaches to rewriting to deal with non-linear structures [Hec06]. There are a variety of graph model transformation tools as compared in Ehrig et. al. [EGdL+05]. A transformation rule matches the model with a pre-condition, which is usually described with a left-hand side pattern that must be found in the input modeland sometimes along with a negative application condition (NAC) [HHT96] that shall not be present (as in triple graph grammar [KÖ5]). The found model part is transformed to what described in the post-condition, which is imposed by the right-hand side pattern.

Figure 2.16 shows the visual representation of a transformation rule that updates value of `P.A`.

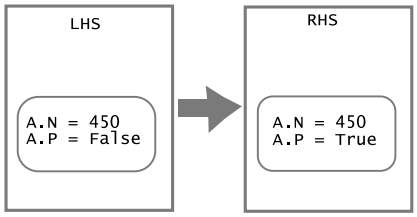


Figure 2.16: Graphical Transformation Rule

The approach is useful and explicit. However, the size of transformation rules may grow fast. In addition, basic model manipulation operations such as Create, Update, Read and Delete [RK09] are yet to be supported in the transformation. It is still a high level abstraction rather than a directly executable action model.

Some model transformation tools improved the transformation by scheduling the rules to specify in which order the rules are applied. The scheduling construct often takes the form of a control flow[5].

Model transformation has been further developed with the introduction of Higher Order Transformation [TJF+09]. It means that transformations taking other transformations as input and/or transformations producing other transformations as output.

The above mentioned execution approaches have proven to be useful in many applications. However,

[5]Tools use graph transformation are, for example, PROGRES [Sch97], AGG [Tae00]and AToM^3 that mentioned in the **Introduction**. The graph transformation modelling tools with rule scheduling include ATL [JK06], FUJABA [GSZ04], GReAT [AKN+06], MoTif [Syr09], VIATRA [VB07] and VMTS [vmt]. OMG has proposed Queries/Views/Transformations (QVT) [qvt09] which ensure a transformation language that queries MOF models and defines transformation is declarative and expresses complete transformations.

these implementations are platform-dependent, specific to domains or hard to modify and update once implemented. These problems lead to the researches in including actions in the root of the metamodelling architecture, the meta-metamodel. Since the meta-metamodel is platform-independent and self-described, the action metamodel, as part of it, is platform-independent as well. At the same time, modelling, instead of hand coding, gives developers much easier models to maintain. Simulating action models and generating complete code is an attractive perspective.

Meta-Metamodel Approach

The following metamodelling approaches add an action metamodel to the meta-metamodel of choice in order to support executability. This type of metamodelling tools facilitate tool-users with semantically rich operations such as simulation, model evolution and execution.

fUML

OMG also notices the importance of including executability in metamodelling languages. They have introduced a beta version foundational UML (fUML) [FUM08]. fUML is a subset of UML. It evolves from UML's modelling static model to describing both static, structural and dynamic, operational behaviour information. fUML owes its executability to an execution model which specifies how UML models are executed in an platform-independent manner.

Different from UML, fUML does not represent everything using graphics, whereas it supports various way of describing model execution. fUML uses a notation similar to Java to describe the "activities", the basic component of actions. "Activity" is the only kind of user defined behaviour supported in fUML. Therefore, all the actions in the execution model must be modelled as activities. Figure 2.17 shows an example of fUML model.

```
// Create a new parameter value for the same parameter as this parameter value, but with
copies of the values of this parameter value.

ParameterValue newValue = new ParameterValue();

newValue.parameter = this.parameter;

ValueList values = this.values;
for (int i = 0; i < values.size(); i++) {
    Value value = values.getValue(i);
    newValue.values.addValue(value.copy());
}

return newValue;
```

Figure 2.17: fUML Example

The fUML execution engine is responsible for model interpretation. It is capable of concurrent execution. The engine consists of three packages:

- Loci of executor (which provides abstraction of executing a fUML model) and locus (which is the physical or virtual computer on which the model is executed),
- Class for UML constructs,
- Common Behaviour (which is the foundation for behavioural semantics).

EP: Events and Properties

Kelsen et al. [KPG08] developed a small behavioural modelling language. This language is based on two main types of elements: **E**vents and **P**roperties (hence the name EP). Additional related elements and OCL code snippets augment these basic elements in order to provide an executable specification of the system.

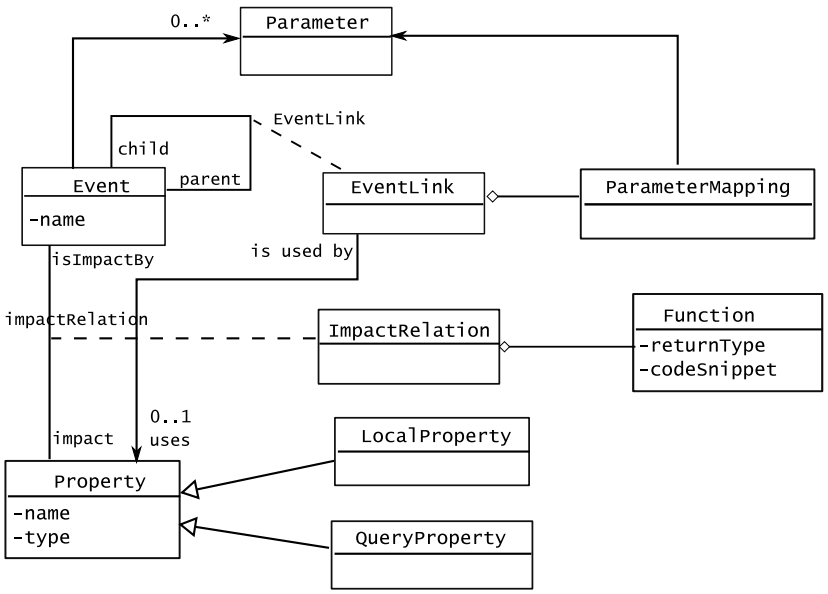


Figure 2.18: EP Behaviour(Action) Metamodel

The structural elements are defined by a modified UML diagram. EP considers a class has attributes and operations. The operations are further divided into two groups: query operations that do not modify state and modifying operations that do. EP adds a component of "events" as modifying operations whose semantics will be detailed in the behavioural model. The EP executable language is as defined in Figure 2.18. While query operations are defined in the structural model via OCL expressions, the definition of the modifier operations is handled by the EP model.

EP has both the abstract syntax and the concrete syntax.

EP's static semantics determines whether an EP model is well-formed. It also defines the dynamic semantics (the actions). At the run time the system state comprises a set of instances, essentially the object graph complying with the class diagram describing the static structure of the system. A state is changed to another by triggering an event on an instance. The value impacted by this event is updated by the OCL expressions attached to the "impact links". EP recursively triggers the events on the target states to execute the model.

Although EP can describe the full specification of the behaviour in a platform-independent manner, EP's use in a real system is restricted because of limited OCL types.

A first prototype supporting EP abstract modelling has been developed based on the DEMOS tool that supports platform-specific executable modelling [KPG08] .

Kermeta

Kermeta [MFJ05] augments MOF with an action metamodel. It uses aspects to weave metamodels of existing metadata languages with precise action specification metamodels. Multiple inheritance defines the behaviour of objects and supports variability. In any case, they believe that the conflict resolution mechanism should be explicitly stated by the programmer.

The action primitives provided in Kermeta includes:

1. Conditionals, loops and blocks.
2. Local variable declarations.
3. Assignment expressions for assigning variables and properties.
4. Literal expressions for primitive types,.

5. Exception handling mechanism allowing exceptions to be raised and caught by rescue blocks. A limited form of lambda expressions corresponds to the implementation of OCL-like iterators such as collect, select or reject on collections.

Kermeta starts from the premise that providing programmers with understandable primitives is of great importance considering many modellers have some programming experience. Kermeta primitives change the syntax of formal methods to what is more familiar to programmers. Using the components mentioned above, modellers are able to build action models in text and these model are thereafter interpreted by the metamodelling tool Kermeta.

XMF

XMF is proposed by Tony Clark et al. [CSW08]. Although their belief on "modelling language is of the same kind with programming language" is arguable, XMF proposes an executable meta-metamodel.

XMF has both abstract syntax and concrete syntax.

XMF is built on the basis of MOF and OCL. MOF supports the standard object-oriented modelling concepts, while an augmented OCL, named XOCL (eXecutable OCL), provides necessary action primitives to enable execution. In combination with a model querying and navigation language, XOCL is a combination of executable primitives and OCL [CSW08].

The extensions to OCL primitives are as below.

- while, find, case, table;
- TypeCase expressions;
- of(), getStructuralFeatureNames(), get(), set() that access and manipulate object properties.

We have introduced several executable meta-metamodels in this section. Considering the primitives of the above meta-metamodels, they all involve control flow, data types and supported operations, and the primitives for modelling such as Create, Update, Read and Delete. Import is also an important operator to enable re-usability.

2.3 Graph Representation

In software engineering, Graph is the common data structure used for internal representation due to its algorithmic power and mathematics foundation [WKR01]. The graphs is an expressive, visual and mathematically precise formalism for modelling of objects (entities) linked by relations; objects are represented by nodes and relations between them by edges. Nodes and edges are commonly typed and attributed [Hec06].

Using graphs makes model transformation, simulation and interoperability easier as the graph eliminates the differences between different formalisms. Graph is useful to standardize the model serialization and therefore ease the task of transporting models between different tools.

There are many different types of graphs, directed and undirected, cyclic and acyclic, etc. to meet different requirements. A tree is considered as a special kind of graph and is widely used. Hierarchical, Attribute, Direct, Typed (HADT) graphs are of particular importance for metamodelling. As illustrated in Figure 2.19, the labelled graph allows for typing of graph nodes; the directed graph allows directed associations; the attributed graph considers graphs are not only distinguished ontologically but also through the properties of each node; the hierarchical graph is the graph containing other graphs. The combination of four graphs satisfactorily covers the features of object-oriented models and metamodelling.

Representing models with hierarchical graphs is useful for multi-paradigm tools. Firstly, many formalisms are hierarchical, such as Petri Net and Statecharts. Having a notation of hierarchy at the kernel level eases the task of modelling such formalisms and also unifies the meaning of hierarchy within the meta-modelling system [Pro05]. Hierarchy allows for modularity, reuse, encapsulation, abstraction and boundary-crossing edges for free.

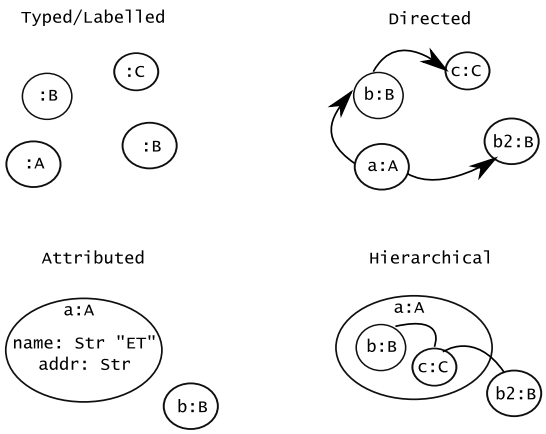


Figure 2.19: Hierarchical, Attributed, Directed, Typed Graphs

There is great diversity in the implementation of graphs. There are graphic libraries in C++, Java and other programming languages; the attributed, typed, directed graph such as ASG and TGraph [ERW08]; the hierarchical, attributed, directed graph such as GraphXML [HHMM00] and GML [Him96]; the hierarchical, attributed, typed, directed graph such as GXL [HWS00] and Himesis [Pro05]. Careful choice of the underlying graph model has significant influence on the compatibility and performance of a metamodelling system.

2.4 AToM3: A Tool for Multi-Formalism and MetaModelling

AToM^3 [dLV02] is the predecessor of AToMPM. It is developed in MSDL at McGill University in close collaboration with Prof. Juan de Lara at the Universidad Autonoma de Madrid, Spain. In this section, we will introduce some important features of AToM^3.

AToM^3 is developed in Python. Figures in Section 1 shows a few screen-shots of the AToM^3 metamodelling environment.

AToM^3 adopts a N-layer metamodelling architecture, where the number of layers is flexible so that modellers can add as many "instance-of" relationships as necessary. AToM^3 usually uses Entity Relationship Model as the meta-metamodel of the system. From a metamodel, AToM^3 generates an environment to visually manipulate (create, edit and analyse) models in the specific domain. AToM^3 has a Button Model to create user interface as shown in Figure 2.20.

Model transformations are performed by graph rewriting. The transformations themselves can thus be declaratively expressed as graph-grammar models [MSD09]. By supporting the transformation, instead of building a completely new formalism, it transforms the new formalism to a known formalism, so the new formalism can use the semantics of the known one.

The main component of AToM^3 is the Processor. It is responsible for loading, saving, creating and manipulating models, as well as generating code. By default, a meta-metamodel is loaded when AToM^3 is invoked. This meta-metamodel allows modelling metamodels to use a graphical notation. The Entity Relationship formalism extended with constraints is available at the meta-metamodel level. The entities must be specified together with their attributes/In the ERM model, it is also possible to specify the graphical appearance of a type of entities in the lower metamodelling layer. This appearance is, in fact, a special kind of generative attribute. We can specify how some semantic attributes are displayed graphically [dLJVM03].

AToM^3 supports automatic code generation, but these generators are not universal but depending on the formalism. Code generation is respectively defined for metamodels. For example, from Statecharts models, a compiler can generate C++, Java and Python code and the documentation.

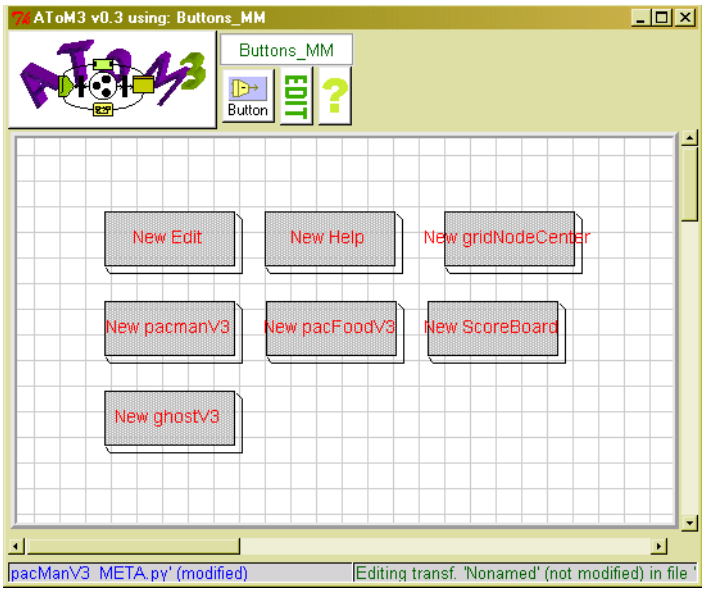


Figure 2.20: Button Model of a Formalism

AToM^3 supports extensive types of properties. Besides the primitives types and the enumeration, in specific formalism such as Class Diagram, it also accepts transformations, Statecharts and class diagrams as attributes of the classes. AToM^3 types are treated as models, and stored as graphs. This implies that specifying customized types is easy and the well developed graph grammars can be constructed to specify operations on types.

ASG: Abstract Syntax Graph

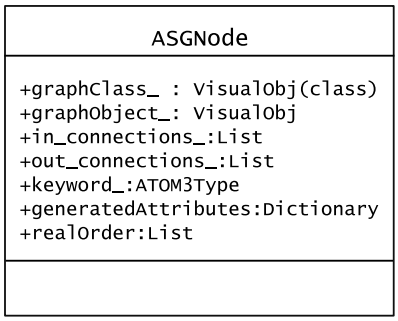


Figure 2.21: AToM^3 Abstract Syntax Graph Definition

In AToM^3, formalisms and models are stored as graphs. ASG (Abstract Syntax Graph) is chosen for the graph model. Every model has a root that is an ASG node. This graph is attributed, directed, but however, not hierarchical. Figure 2.21 shows the definition of an ASG Node.

When saving the models, the AToM^3 Processor generates a Python class for each entity type along with one Python file for the ASG graph. The former classes are called to instantiate these entities in the lower metamodelling layer. The latter file is responsible for storing entities as the nodes of the graph, the global constraints, and connectors that link the graphic entities.

```
self.obj25=pacmanV3(self)
    self.obj25.isGraphObjectVisual = True

    if(hasattr(self.obj25, '_setHierarchicalLink')):
      self.obj25._setHierarchicalLink(False)

    self.obj25.graphClass_= graph_pacmanV3
    if self.genGraphics:
       new_obj = graph_pacmanV3(380.0,180.0,self.obj25)
       new_obj.DrawObject(self.UMLmodel)
       self.UMLmodel.addtag_withtag("pacmanV3", new_obj.tag)
       new_obj.layConstraints = dict() # Graphical Layout Constraints
       new_obj.layConstraints['scale'] = [1.0, 1.0]
    else: new_obj = None
    self.obj25.graphObject_ = new_obj

    # Add node to the root: rootNode
    rootNode.addNode(self.obj25)
    self.globalAndLocalPostcondition(self.obj25, rootNode)
    self.obj25.postAction( rootNode.CREATE )
```

The above code shows an AToM^3 class object represented in ASG. After constructing the ASG node `obj25` for the entity type `pacmanV3`, this object is drawn and positioned. At last, `obj25` is added to the root node of this model and rendered.

One drawback of ASG is that even for non-graphical formalisms, one must devise a graphical representation. Consequently, the abstract syntax and concrete syntax are not clearly distinguished.

Transformation and Executability

Model transformation refers to the (automatic) process of converting, translating or modifying a model in a given formalism, into another model that might or might not be in the same formalism.

To do this, AToM^3 specifies model transformations graphically (based on the ASG). As shown in Figure 2.22, to set up a transformation, the metamodel of source model, the metamodel of the target model and the metamodel for transformation should be loaded beforehand. The designer creates transformation rules which consist of priority, the precondition (LHS graph), the constraints associate with ASG nodes (as in Figure 2.23), and the result of transformation (RHS graph). Sometimes an action is inserted when modifications are required of an ASG node. The pattern matching engine explores the current model for LHS graphical pattern and executes the transformation rule to one of the search results. Repeating until no matching can be found, the source model is transformed to the target model.

The inconvenience is that AToM^3 does not have a neutral, general-purposed action metamodel. AToM^3 models are compile to Python modules for simulation. The actions and constraints of transformation rules are also coded in Python. Therefore the developer has to have the knowledge of both Python and how entities are organized in AToM^3.

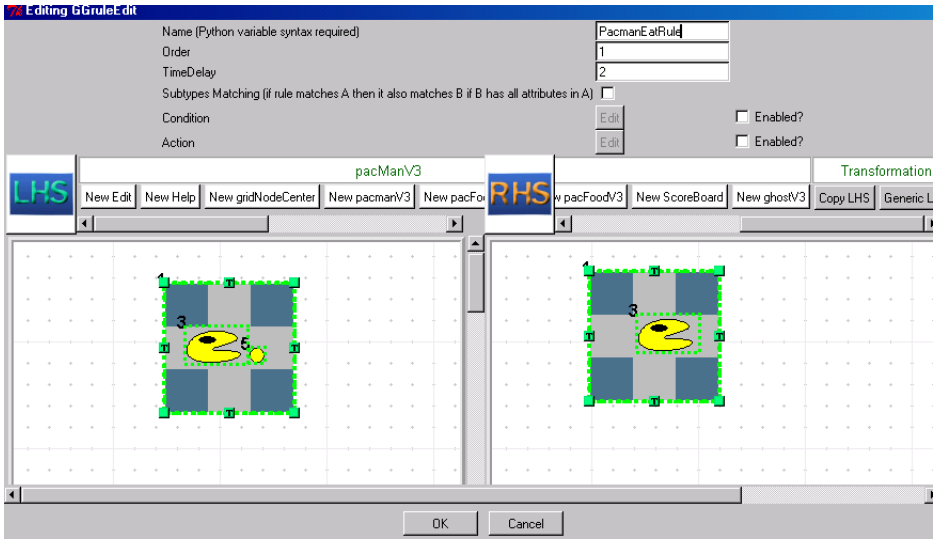


Figure 2.22: Define a Graph Transformation Rule in AToM3

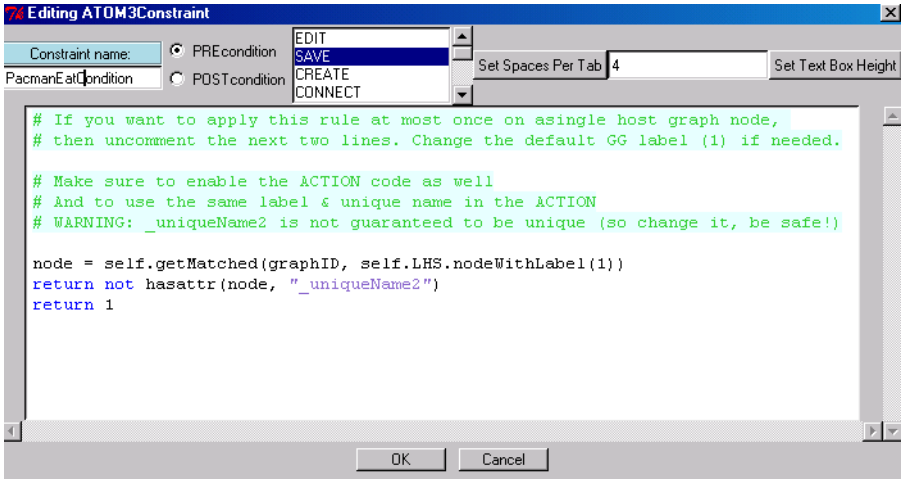


Figure 2.23: Define a Constraint in Model Transformation in AToM3

This code-generating tool, developed in Python, relies on graph grammars and metamodelling techniques and supports hierarchical modelling. AToM^3 has the following features that will be changed in AToMPM.

1. AToM^3 uses Python to define actions. It does not have neutral general-purpose action language.
2. AToM^3 mixes the concrete syntax and abstract syntax, which makes models hard to maintain and updates.
3. AToM^3's ASG graph node is not hierarchical, which limits the tools metamodelling and transformation ability.

It is also worth to extend the tool to allow collaborative modelling and version control in the future.

2.5 Summary

As introduced in this chapter, we have walked through all the relative concepts of metamodelling. We have explained the common composition of metamodels;various metamodelling frameworks; existing meta-metamodel; current approaches to enforce constraints; and current attempts to enable executability. Also, many metamodelling tools have been developed in various domains. Despite their acceptance in respective domains, there are some short-comings that limit their success as a conceptually elegant metamodelling approach. The characteristics of a conceptually elegant tool, as we propose, are,

- there is a meta-metamodel as the basis of all the formalisms and metamodels being used;
- the metamodelling architecture is closed;
- the tool supports model executability from the very beginning, the meta-metamodel;
- the tool is platform- and language-independent.

3

Two-Dimensional Metamodelling Architecture and ArkM3

We have discussed in the last chapter the challenges of the existing metamodelling technologies. They are the need for a cleanly classified metamodelling architecture and the need for a general and universal meta-metamodel that describes all the aspects including executability and constraints.

The work in this project will be used in the AToMPM (A Tool for Multi-Paradigm Metamodelling) project in the Modelling, Simulation and Design Lab at McGill University. Our contributions in this work are listed below.

1. Design a two-dimensional metamodelling architecture that leads to general-purpose, comprehensive and bootstrapped metamodelling tools. One dimension describes the logical classification of the models and the second the physical classification.
2. Propose a universal hierarchical space, the metaverse, for model storage and model management in a modelling environment.
3. Design a general-purpose, self-describable, executable meta-metamodel—ArkM3 (**A**ToMPM **r**eusable **k**ernel **M**eta-**M**eta**M**odel).
4. Add an Action Language to ArkM3, so as to describe the actions and constraints through out the modelling hierarchy.
5. Adapt the hierarchical, attributed, directed, typed (HADT) Graph Himesis, as the metamodel in the physical dimension.
6. Define a strict mapping between the ArkM3 models and the Himesis models.
7. Implement the **A**ToMPM **r**eusable **k**ernel (Ark) to validate the above design.

Chapter Organization. Section 3.1 talks about contribution point 1; Section 3.2 describes the metaverse at contribution point 2; Section 3.3 describes the meta-metamodel design decision at contribution point 3; Section 3.3.3 describes the action language at contribution point 4; The remaining contributions will be explained in the next Chapter 4.

3.1 Metamodel Architecture

A good metamodelling architecture is the basis of well-defined metamodelling tools. We believe that a well-defined metamodelling tool should be completely platform-independent which indicates that the metamodelling architecture can be bootstrapped. A well-defined metamodelling tool should be built in a modular manner so that tool development and distribution do not interfere with each other. Therefore, to build a tool that is bootstrapped, comprehensive, easily maintained, we propose a two-dimensional architecture.

The AToMPM two-dimensional architecture comprises a logical dimension and a physical dimension. In each dimension, models are organized in layers, which is inspired by the MOF Four-Layered Metamodelling Architecture. At the same time, in both dimensions, we adhere to the strict metamodelling.

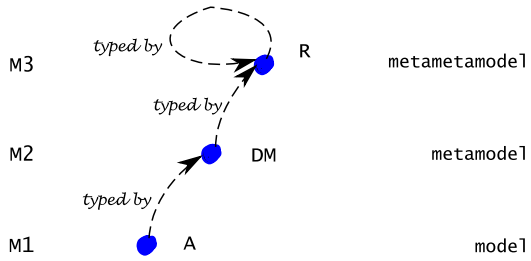


Figure 3.1: AToMPM Two-Dimensional Metamodelling Architecture: the Logical Dimension

The logical dimension describes the abstract syntax of models. It identifies the model layer, metamodel layer and meta-metamodel layer as illustrated in Figure 3.1 which correspond to M1, M2 and M3 meta-layer in MOF respectively (M0 for user object in MOF is out of the scope of the metamodelling tool). The root of the logical dimension metamodelling hierarchy is the meta-metamodel. It is a self-described meta-metamodel typed by itself. Models on each level are strictly typed by the immediate higher level metamodel or metamodels. Three layers are usually enough for modelling, but the number of middle layers is not rigid. As long as the models adhere to the strict metamodelling principle, modellers can add as many layers between the metamodel layer and the model layer as necessary.

AToMPM allows a model to be typed by multiple logical metamodels. To adhere to the strict metamodelling, instead of potency, we allow importing the required parts. Once imported, these parts become a part of the importing model. They are treated the same as other parts of this metamodel. Although the size of the metamodel is expanded, the benefit of importation can not be ignored. The metamodel contains all necessary information at one place. The complexity to maintain the consistency between models and metamodels is reduced. The users should be free to explicitly instruct if the imported features should be updated along with the source metamodels In the future.

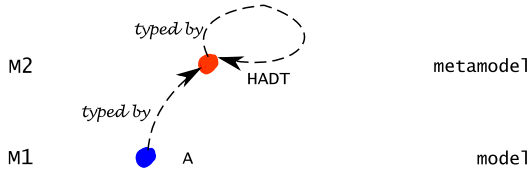


Figure 3.2: AToMPM Two-Dimensional Metamodelling Architecture: the Physical Dimension

The physical dimension describes the internal representation of a metamodelling tool. HADT graphs support the scoping of models, packages and encapsulation for free. At the same time, they also allows for unique paths for all the models elements. So far, two layers is enough in the physical dimension, which are the graph model (A) and graph metamodel ($HADT$) as in Figure 3.2.

Eventually, we have a two-dimensional representation of models as that in Figure 3.3.Logically, A is defined by DM and DM by R, and physically A is typed be $HADT$. A is a model whose abstract syntax is defined by metamodel DM, and physical implementation is defined by $HADT$. The modellers are aware of the hierarchy in the logical dimension, while the tools operate on the graph models. As a result, the metamodelling tools can easily separate the responsibility of tool developers and that of modellers.

To integrate the two dimensions, a precise mapping is required. All the models in the logical dimension, regardless of on which meta-layer in the metamodelling hierarchy, have projections onto the physical dimension. As we can see, both R and DM are typed by $HADT$.

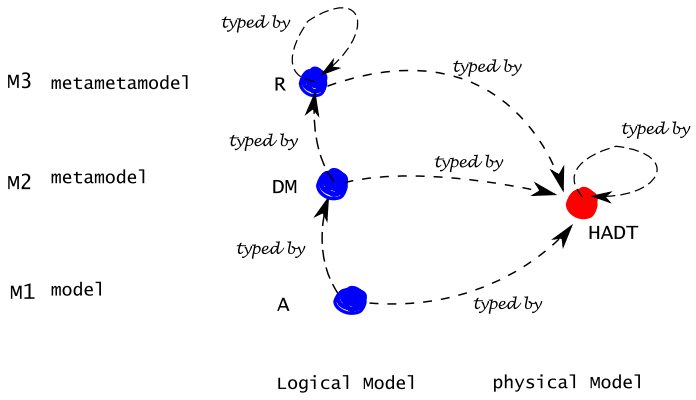


Figure 3.3: AToMPM Two-Dimensional Metamodelling Architecture

3.2 Universal Hierarchical Modelling Space—the Metaverse

The metamodelling tools need to store and manage models in an efficient and organized manner. One motivation is to conceptually draw a clear boundary between the modelling world and a computer or a server, which are actually conceptually distinct: the former is the set of models, and the latter is media to store models. Another motivation is to facilitate collaborative programming and modelling. The run-time may store the models in local computers and cross the internet. Multiple concurrent users may access to a model. Modellers may refer to other models, import model components from co-workers, and even request remote resources across the Internet.

This adds to the metamodelling tool requirements. First, metamodelling contains models that was modelled, being modelled and to be modelled. To a metamodelling tool, this enables automatic linking and also allows for modelling resource to spread over different physical locations. Second, models are organised in user accounts instead of by folders or by meta layers. Third, models are stored hierarchically and the access is authorized. Last but not the least, models have a globally unique ID to assists the importation, reference and internet request.

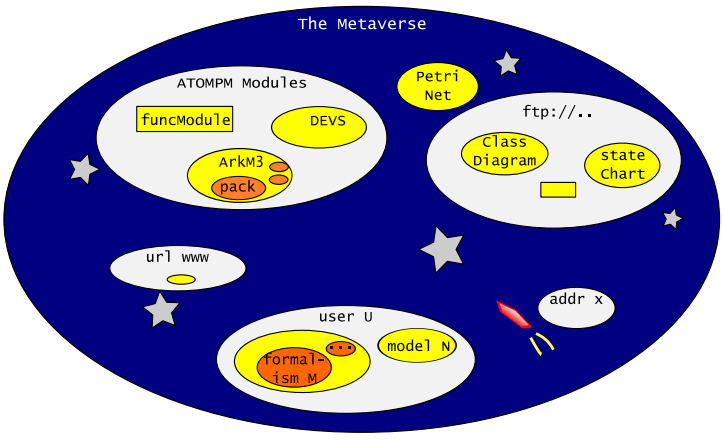


Figure 3.4: The Metaverse

We propose a universal hierarchical metamodelling space called 'Metaverse'—the combination of '**META**modelling' and 'uni**VERSE**'. Let us use the real universe as a analogy. As the universe, the metaverse is the universal storage space for all the known and unknown, existing and future models. As shown in Figure 3.4, the metaverse does not distinguish models by their metamodelling layers but treats the all the models

equally. The models may be clustered by user (user group), url address and so on. We consider that compiled binaries may be modelled as well, hence binaries is saved in the metaverse. They are equally accessed as the models. The metaverse overlooks the content of metamodelling tool and is able to assist our modelling activities.

AToMPM takes a subsection of the metaverse, as if the known part of the universe, which we can call a "work space". Ark is a subsection AToMPM work space. It contains the meta-metamodel ArkM3 and kernel function modules.

The size of work space is dynamic. A work space expands as more models are used. When the required model is outside the work space, the user need to either add it to the work space or provides a reference. It is like to discover an unknown constellation in the universe, our view of the universe grows via either observing a new star or calculation from the related stars. An external resource is added to the workspace as shown in Figure 3.5.

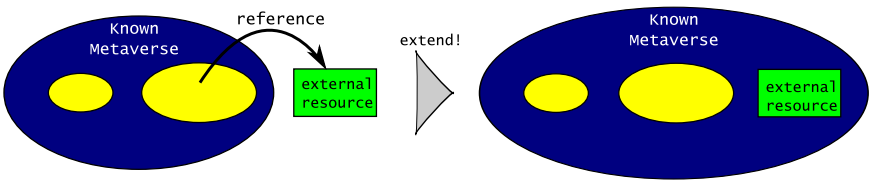


Figure 3.5: Extend the Work Space

As for the implementation of metaverse in AToMPM, we adopts the same HADT graph that is used to represent models. The metaverse's hierarchical structure is well supported by HADT graph. Furthermore, using the same graph structure allows the model and groups of models to be treated uniformly.

3.3 ArkM3: the AToMPM reusable kernel Meta-metamodel

Instead of using AToM^3's Entity Relationship formalism as the meta-metamodel, AToMPM has ArkM3, an executable, self-described, general-purposed meta-metamodel as the root of the logical dimension. ArkM3 comprises of the Object Package, the Data Type and Value Package and the Action Language Package, as shown in Figure 3.6. These packages collaborate to describe all the information of domain specific models.

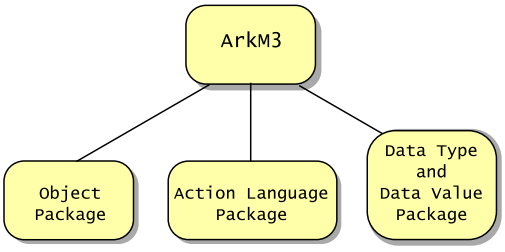


Figure 3.6: ArkM3 Packages

Section Organization. As for each package, we will introduce their abstract syntax, constraints and operations, and the semantics respectively. Note that the concrete syntax is not part of ArkM3, although mapping to Himesis graph node can be considered as the concrete syntax in some aspects. The mapping between the logical models and the physical models will be explained in the Chapter 4, Section 4.3.

3.3.1 The Object Package

The Object Package is based on OMG's standard EMOF (Section 2.1.3). It is because that EMOF is the minimal set of UML, so that using EMOF brings simplicity to the meta-metamodel and helps minimizing the AToMPM kernel. It is also because EMOF is a well-recognized standard, so that using EMOF thereby lowers the entry to ArkM3. Another reason is that the MOF standard is widely recognized in the modelling world and the many tools have been developed to support MOF, so that using MOF-like models helps to save the development time and allows models to immigrate to and from other tools with least effort.

ArkM3 Object Package (Figure 3.7and 3.8) contains the definitions of package, class, association and composition.

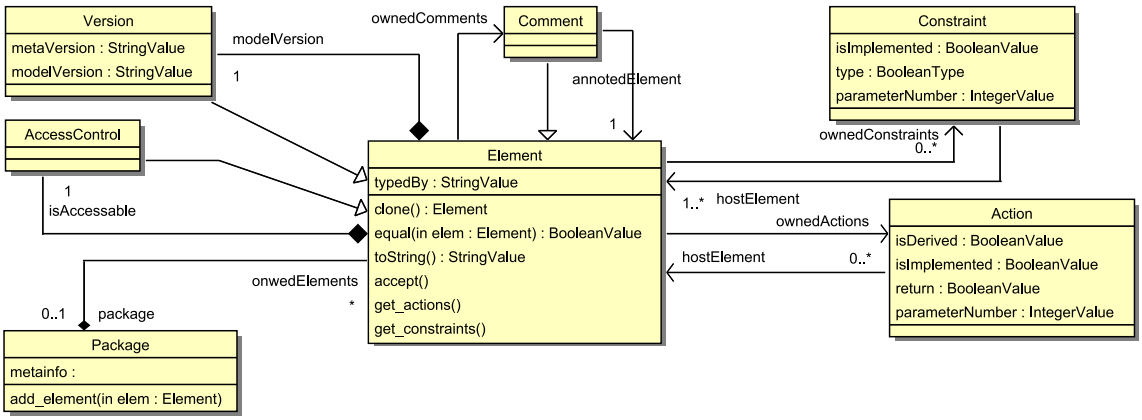


Figure 3.7: Meta-Metamodel of Object Package: the Element

In ArkM3, as shown in Figure 3.7, everything is a subclass of *Element*, the constituent of models. *Element* is an abstract class and it has no super classes. It describes the common properties of all the model objects. The attribute *typedBy* indicates that a model element should always be typed by a metamodel element. *Element* have operations to clone itself, compare to others, reflect the content, implement the accept visitor method. *Element* associates with $Version$, *Comment* and *AccessControl*, which respectively implies that modellers can specify the model version, authorize access privileges and comment the elements. *Element* also associates with *Actions* and *Constraints*, which implies that all elements are eligible to have constraints and executable actions.

More specifically, we identify $TypedElement$, $NamedElement$ and $MultiplicityElement$ to represent an element having type, name and multiplicity respectively. We also define *Package*, which inherits from $NamedElement$, contains either elements or other packages. We enable hierarchy in ArkM3 models using *Package*. It is a mechanism of grouping related model elements together in order to manage complexity and facilitate reuse. It enables combining new or reusable metamodelling features to create extended modelling languages. *Package* provides visibility for free. Package merge combines the features of the merged packages with the merging package to define new integrated capabilities.

Figure 3.8 shows the rest of the Object Package. The definition and semantics are similar to that of EMOF described in Section 2.1.3, though there still are some differences. A $TypedElement$ has $Type$, while $Type$ is also a $NamedElement$. Its type is modelled by $TypeType$, a subclass of $Type$. ArkM3 defines both $DataValue$ that inherits from $TypedElement$, and $DataType$ that inherits from $Type$. The difference between $DataValue$ and $DataType$ will be explained in Section 3.3.2.

Class have attributes and operations. An abstract class can not be instantiated. *Class* defines getters and setters. The difference between the actions and the operations is that an action is implemented at the current level of metamodelling, while operations is to be implemented when this model is instantiated. We distinguish the inheritance relationship and the generalization relation of classes, although this is

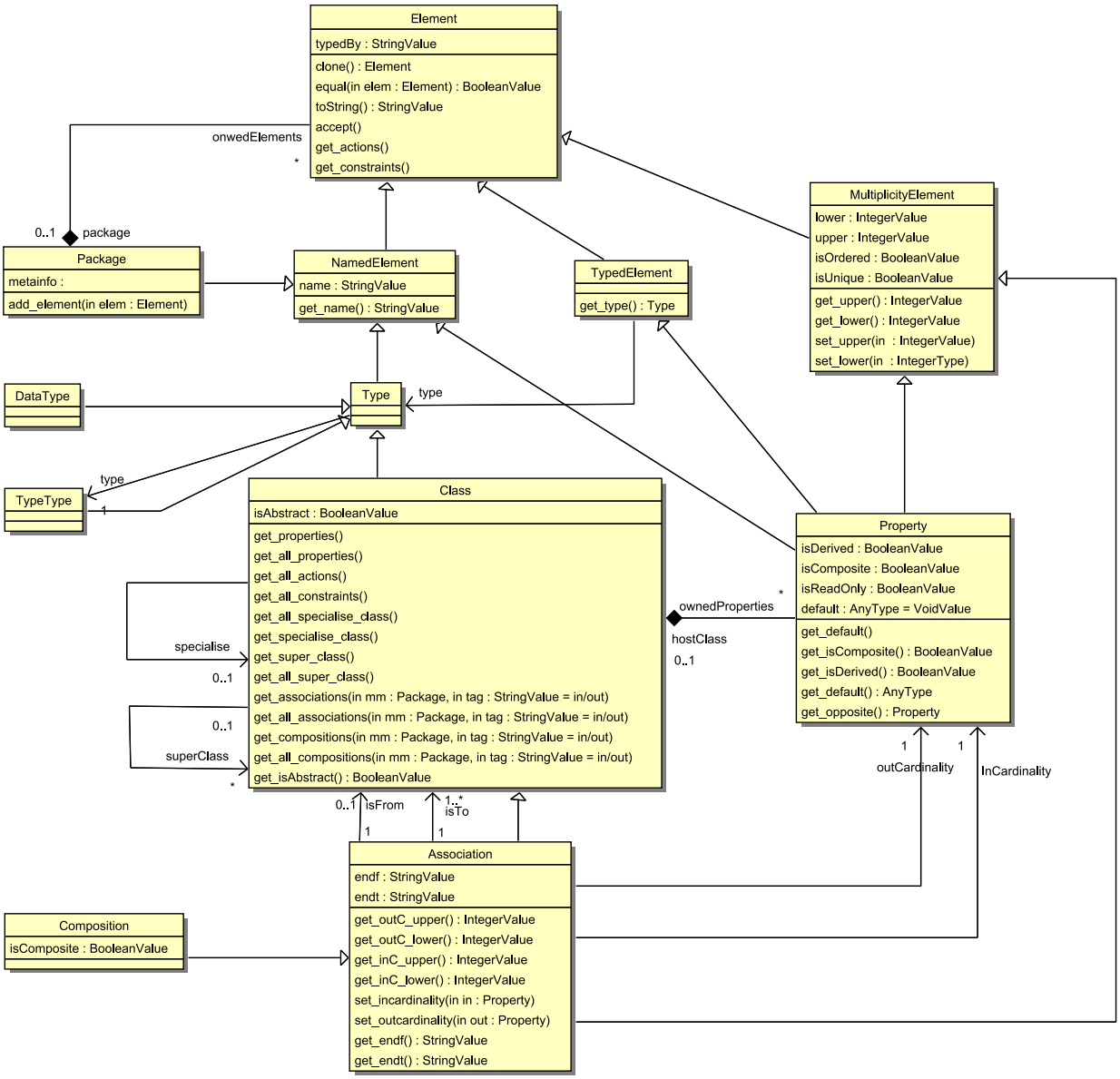


Figure 3.8: Meta-Metamodel of Object Package: Package, Class and Association

not a usual approach. As discussed in [KÖ06], generalization and inheritance are conceptually different. Distinguishing the two will bring clarity to the metamodelling process. Class also have corresponding operations to access inherited attributes and methods.

ArkM3 supports multiple inheritance. If conflict occurs, the system would try to resolve the conflict in the first place, otherwise it invokes an exception in the tool level and invites modellers to resolve the conflict.

There is only one generalization for a class at most, since the subsets of a class is not likely the subset of another.

Association inherits from *Class* and in this way it also has the ability to derive and specialize. The associations can have subclass and even be connected by associations. Thus, this inheritance is of importance for higher order transformation. *isFrom* and *isTo* are respectively the source and the destination of an association. One association can only have one source. An association can have multiple destinations, the multiplicity of *isTo* is set as "1..*".

Composition inherits from *Association*. The inwards and outwards cardinalities of *Association* and *Composition* are properties.

Generally speaking, Object Package defines the classes for structural metamodelling. It gets ready for the excutability by adding plugs to action models and constraint models in the root class *Element*. Inheritance and generalization are distinguished to clarify two different class relationships. *Association* inherits from *Class* so as to facilitate the higher order transformation.

3.3.2 The Data Type and Data Value Package

Along with the Object Package, we define data types and data values in the Data Type and Value Package. In ArkM3, everything is an object. Not only classes, but also an integer and a data type can be instantiated.

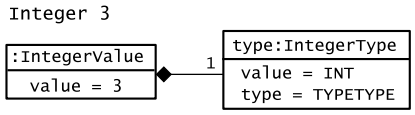


Figure 3.9: Data Value Example: An Instance of $IntegerValue$

To meet the philosophy of "modelling everything explicitly", We modify the common type system. Instead of having a data type class representing values and implicitly the type of these values, ArkM3 explicitly defines data's type. AToM^3 has both $DataType$ and $DataValue$ classes. For each instance of a specific value, there exists an explicit data type. $DataValue$ class—"data type" as commonly called—comprises of a $DataType$ instance called named "type" explicitly states the type. For example, as illustrated in Figure 3.9, an integer 3 in ArkM3 will be represented as an instance of $IntegerValue$ with value 3, and the type of this $IntegerValue$ instance is "INTEGER", the instance of $IntegerType$.

The ArkM3 Data Value package, as shown in Figure 3.10, provides primitive data values and enumeration. $EnumerationValue$ contains ordered $StringValue$ or $StringLiteral$. The package also inluded and collection—set, sequence and dictionary, for user's convenience, as listed in Table 3.1.

Shown in Figure 3.11 is the Data Type Package. Each class in this package corresponds to a class in the Data Value Package. The type classes are singletons. We have a special type of $AnyType$, which implies that a data value's type is not determined. Instances of $VoidValue$ have the type "ANY".

Comparing with AToM^3, ArkM3 Data Type and Value Package supports fewer kinds of data values, as ArkM3 only intends to provide the essential minimum. AToM^3 does not explicitly define data types, whereas ArkM3 does.

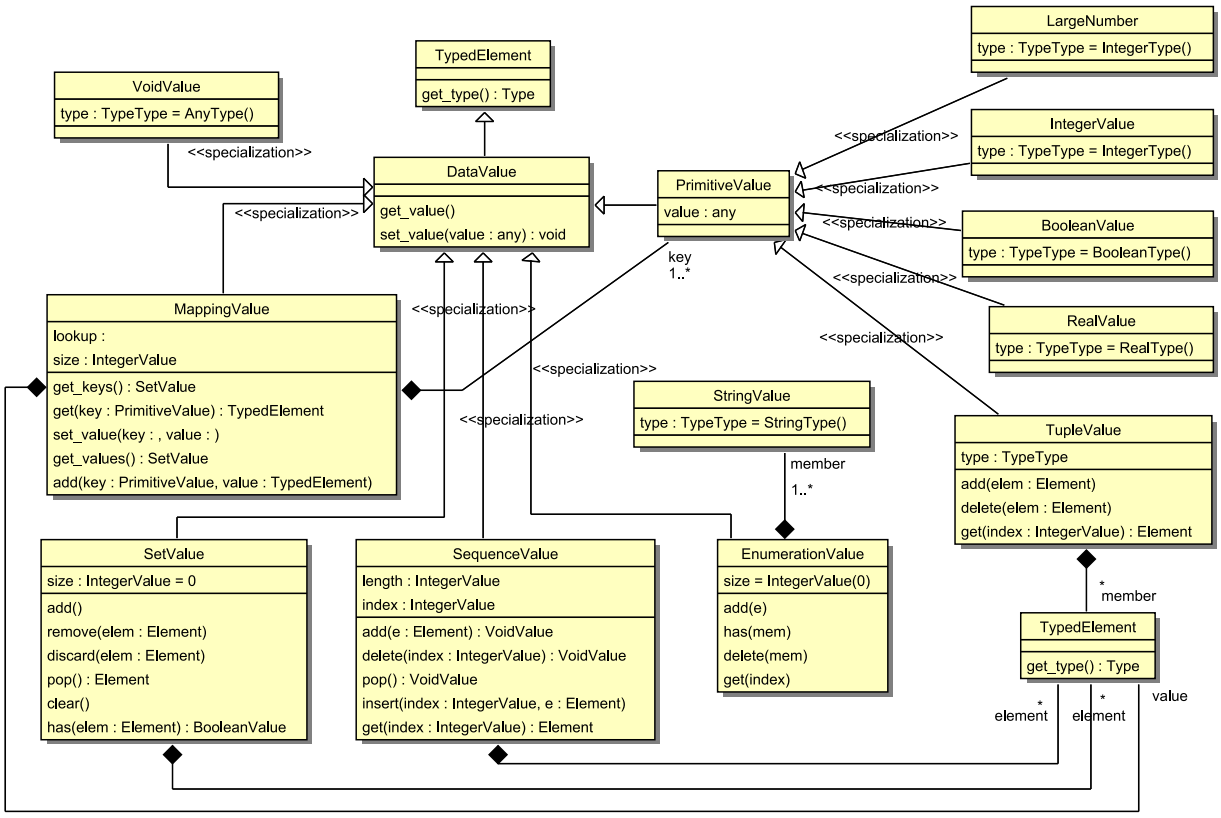


Figure 3.10: Meta-Metamodel of Data Value Package

Table 3.1: ArkM3 Collections

Data Structure	*Orderliness*	*Uniqueness*	*Type*	*Operations*
SequenceValue	True	False	SequenceType	get, add, delete, insert, pop
SetValue	False	True	SetType	has, add, remove, pop
MappingValue	False	key is Unique	MappingType	has, get, get_keys, get_values, add, delete, set_value

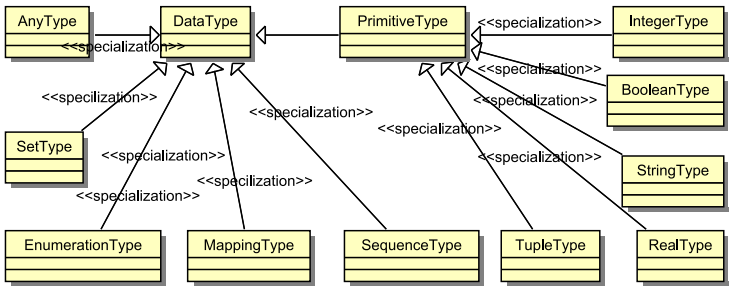


Figure 3.11: Meta-Metamodel of Data Type Package

3.3.3 The Action Language Package

The Action Language Package models a neutral, general-purposed action language. The Action Language Package describes the syntax, semantics, constraints and execution of operational behaviour. The constraint is a subset of the action, therefore this package can also model constraints. With Action Language Package, we can make domain models well-formed, complete, executable and platform-independent.

This package contains the following components,

- *Action* and *Constraint*;
- statements;
- expressions;
- literals;
- primitive operators.

The principle of choosing components are: 1. the number of primitives should be minimal; 2. the action language should contain enough concepts of object-oriented programming languages; 3. basic transformation operations is needed. Thus, the first criterion is whether the action elements can be implemented by the others; the second criterion is whether the action elements are used frequently (frequently used elements should be more efficient). The selection may keep improving over time as the experience accumulates. As mentioned in Section 2.2, other efforts towards implementation-independent action languages are good reference to our work.

The Literal Package

Literal represents the literals of data values. *Literal* is used when data value can not be directly used in the models. For example, when we want to have an integer as a member of a Enumeration, the integer is converted to an integer literal. It is an abstract class. Different types of literals specialize the *Literal* class. *EnumerationLiteral*, *SequenceLiteral*, *SetLiteral*, and *MappingLiteral* are composed of $TupleValue$. The $TupleValue$ instance contains two elements, the first of which is the reference to the respective literal and the second is the index. The literal package is shown in Figure 3.12.

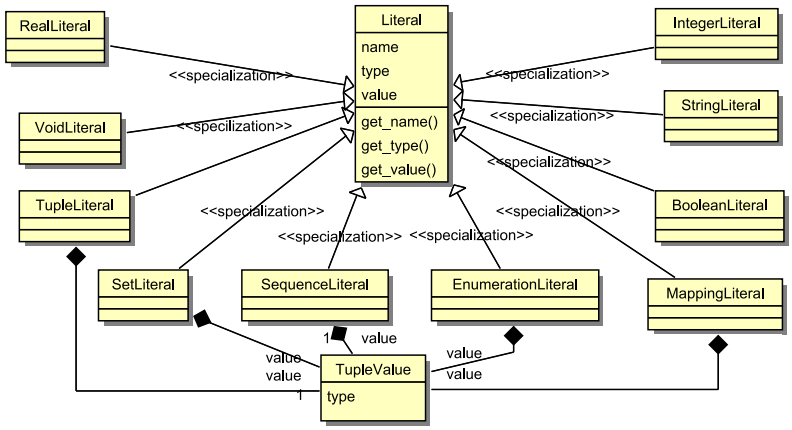


Figure 3.12: Meta-Metamodel of Literals

The way to interpret a literal is coded rather than modelled. The relation between the literals and the *Literal* class is specialization, because these subclass literals does not develop the general class but rather restricts it.

Action and Constraint

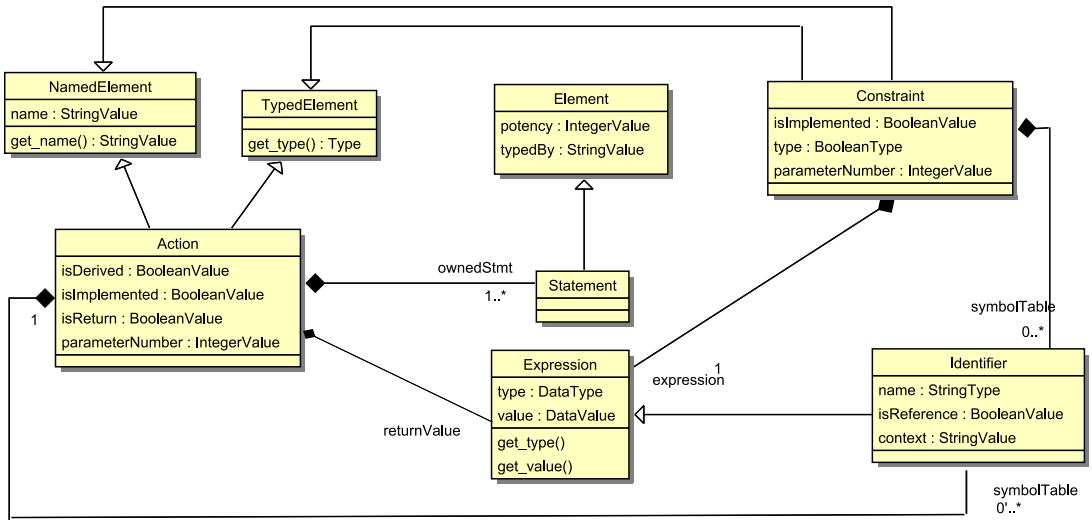


Figure 3.13: Meta-Metamodel of Action Language Package: Action and Constraint

Action and *Constraint* are the roots of an action model and a constraint model respectively. As shown in Figure 3.13, both of them are named and typed. An action models the operational behaviour that may have an effect on the host model, while a constraint is a querying model to examine model status rather than change model status .

An *Action* comprises of *Statements*. It uses a symbol table to maintain variables and a Boolean expression indicating if there is a return value. If a return value is required, a slot in the *Action* instances stores that value. *Constraint* contains one Boolean expression and returns a Boolean value.

Statements

ArkM3 provides notations for control flow such as branches, loops, expression sequences, return statement and declaration. These classes specialize *Statement*, which is an *Element*.

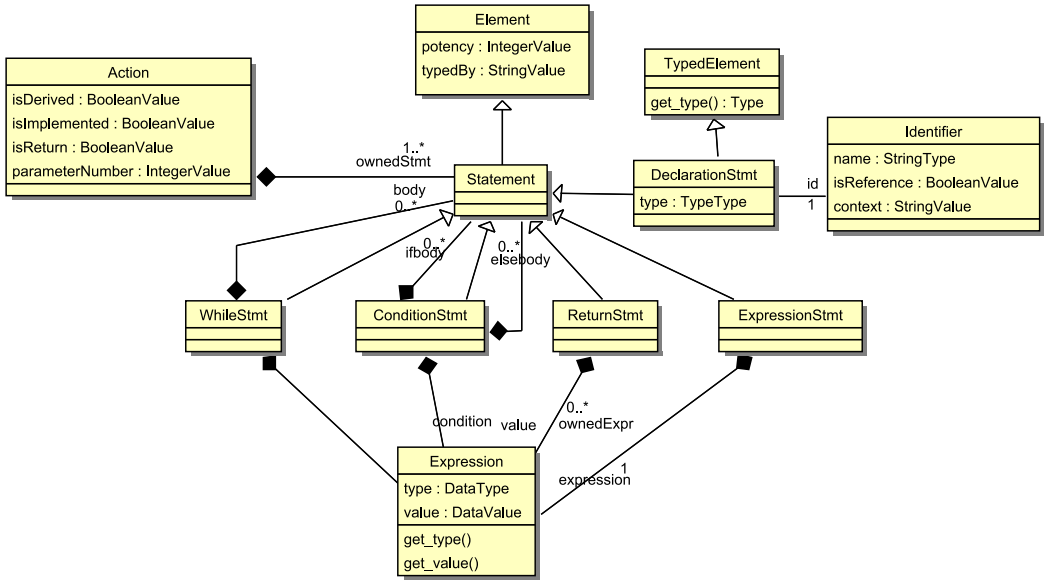


Figure 3.14: Meta-Metamodel of Action Language Package: Statements

Statement	*attributes and compositions*	*Implementation*	*constraint*	*result*
Expressions	expr, ... expr	traverse the content in order.	n/a	n/a
Conditional	expr, stat1, stat2	if expr is true execute stat1 otherwise stat2.	expr value is boolean	n/a
Loop	var, init, term, stat	initialise var, execute stat and update var, iterate until term expression is true.	var is identifier, init and term are expressions	n/a
Declaration	id, type	be suspended because of dynamic typing.	n/a	n/a
Return	expr	return expr's value or reference to obj referred by expr.	n/a	expr's value/object

Expressions

Figure 3.15 shows the class diagram of the expressions.

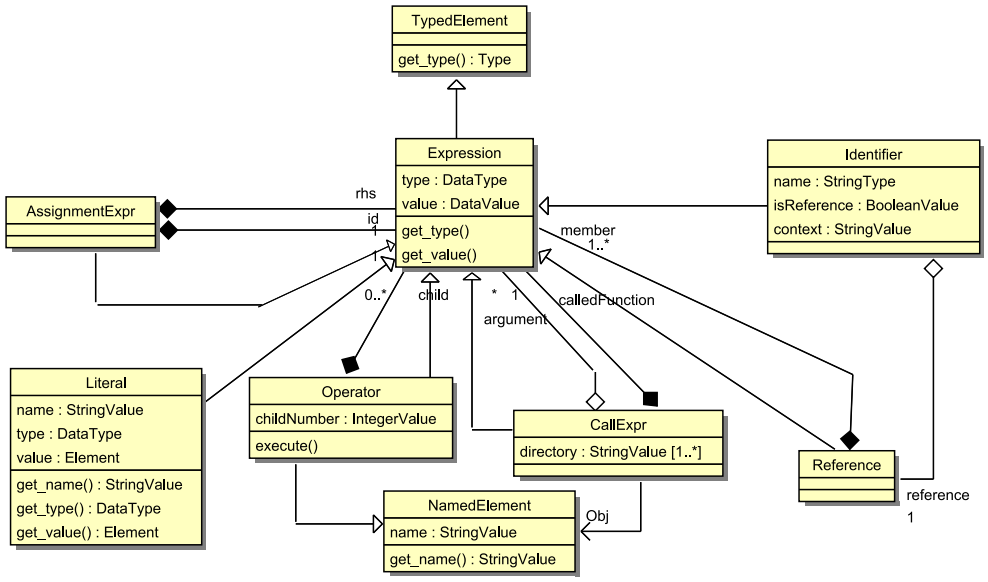


Figure 3.15: Meta-Metamodel of Action Language Package: Expression

Expression	*Attributes and Compositions*	*Constraint*
identifier	name, type, value, isReference, offset	if isRreference is true, it can contain a reference
call	module, called, dir	module can be either model or compiled classes
reference	origin, sequence of strings, value, type	n/a

The Operator Package

Operator is a subclass of *Expression*. They are the most basic elements in the Action Language. These primitives are frequently used so that the efficiency becomes important. Since hard coded primitives are usually faster than those by models, instead of modelled by other operators, the chosen operators shall be implemented by tool developers using some programming languages. Defining an operator as primitive will allow the tool developer to rewrite it when the system is ported to other platforms. These operators are fundamental in describing actions and model transformation.

Figure 3.16shows the complete Operator package.

Arithmetic Operators, Comparative Operators, Boolean Operators and Type Conversion Operators

These operators are designed to represent basic operations, such as arithmetic operators, Boolean operators, comparison operators and type conversion operators. These operators are common in programming languages. Since we want to design an action language that has minimal dependency on the implementation, we ought to have the smallest possible set of fine grained actions.

These basic primitive operators are implemented by directly applying the corresponding Python operators on the operands, such as the operators in the following table.

Operator	*Operands (in order)*	*Implementation*	*example /meaning*	*constraint*

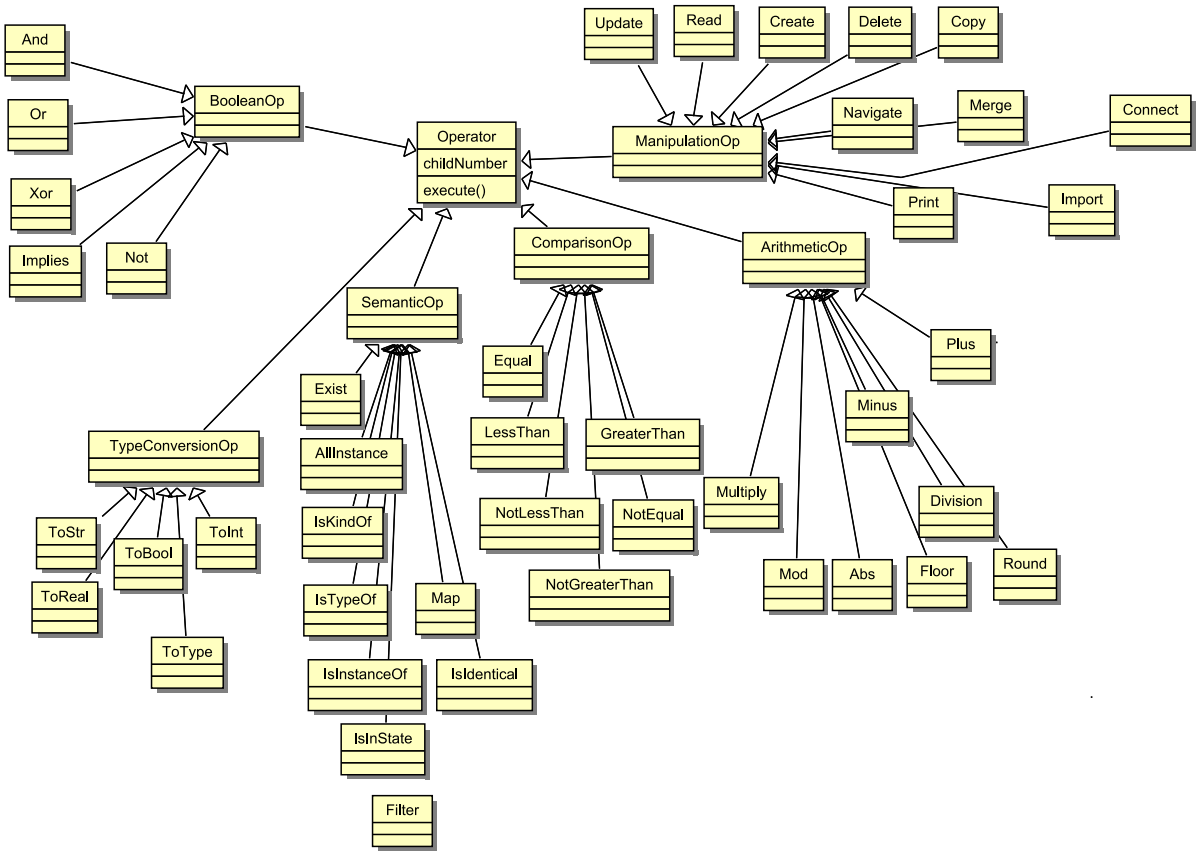


Figure 3.16: Meta-Metamodel of Operators

arithmetic ops				
add	operand1, operand2, ...	_+_	1+2 => 3	op type: int, real or string op #: infinite
division	dividend, divider	_/_	2.0/5.0 = 0.4	op type: int or real op #: 2
boolean ops				
and	operand1, operand2, ...	_&&_	false && false => false	op type: bool op #: 2
type conversion op				
2int	operand	int(_)	int("a") => 96	op type: string, real or bool op #: 1
comparison ops				
lessthan	operand1, operand2	_<_	4<3 => false	op type: int or real op #: 2

Collection Operators

These primitives are used frequently to handle a collection of model elements. It includes *mapping* that applies a common action on a set of objects; $filter$ that scans a set and returns those that satisfy the criteria; $allInstanceOf$ that returns all the object of a certain type.

Operator	*Operands (in order)*	*example/meaning*	*constraint*	*result*
semantic ops (*obj refer to a model element, model and metamodel refer to the entire definition*)				
allInstance	type, model	get all the instances of a certain type	type is string or a model	set of objects
exist	obj, model	n/a	obj is a string or a model	true/false
isKindOf	obj, metaobj	if a model is defined by metaobj or its subclasses	n/a	true/false
isInstanceOf	obj, metaobj	if a model is defined by metaobj	n/a	true/false
isIdentical	obj1, obj2	if obj1 and obj2 refer to the same object	n/a	true/false
mapping	expr, set of objs	apply expr to each element in obj set,	n/a	none
filter	expr, set of objs	select the obj that satisfy the expr	result of expr is bool	set of obj

Model Manipulation Primitives

We also include some operators especially to manipulate models. We use *copy* to duplicate a model element and *import* to merge a model from an external source. *merge* is to integrate different models and can be considered as a model transformation process.

We have included some operators that defined especially because of the use of Himesis hierarchical graph nodes, such as *connect* and *print*. *connect* connects two Himesis graph nodes. *print* pretty prints the model content. Nevertheless, strictly speaking, thses two operators should not be part of the primitives.

Some important primitives are provided for frequently used operations. The meaning and the composition of these operators are as below.

Operator	*Operands (in order)*	*example/meaning*	*constraint*	*result*
manipulation ops *(obj refer to a model element(ArkM3 values and types are also model elements), model and metamodel refer to the entire definition)*				
copy	obj	duplicate obj	n/a	obj
update /assignment	var, expr	assign the result of expr to var	var and expr should be same type, otherwise the var type is changed according to expr	value of expr
delete	obj	delete obj	n/a	n/a
read	obj	value of obj	n/a	value of obj
navigate	obj, path	the object refered by path from obj	path is a string separated by periods	n/a
merge	model1, model2	merge two models	n/a	model
import	model1, models	import model2 to model1	n/a	model1
connect	node1, node2	connect to nodes in a Himesis model	this is especially for **Himesis** graph models	None
print	graph	output the graph content in console	this is especially for **Himesis** graph models	strings

Transformation Primitives

According to the discussion in [MG05], there are four basic required operations for the model transformation. These operations are called as CRUD, which are Create, Read, Update and Delete.

- *read*, reflect the content of the graph node and return the value of a node;
- *update*, equivalent to assigning a new value to certain model element, so that it is implemented in the way of *assignment* expression;
- *delete*, delete an graph node and all the nodes in it;
- *create*, if the metamodel exists, create an model according to its definition. It calls the Create Visitor in function package.

These operations can be applied on models and transformations. With these four operations, the modellers can define any transformation. We adopt CRUD as primitive operators. There is overlap between transformation primitives and other operators—for example, *update* can be implemented by *assignment*—but it does not affect the syntax itself.

In this section, we introduced the meta-metamodel of AToMPM. ArkM3 is the root of the logical metamodelling hierarchy. It is self-describable so that we can bootstrap the metamodelling tool. ArkM3 comprises of three packages—the Object Package, the Data Type and Value Package and the Action Language Package. These packages help to describe every aspects of a model.

4

Ark: AToMPM reusable kernel

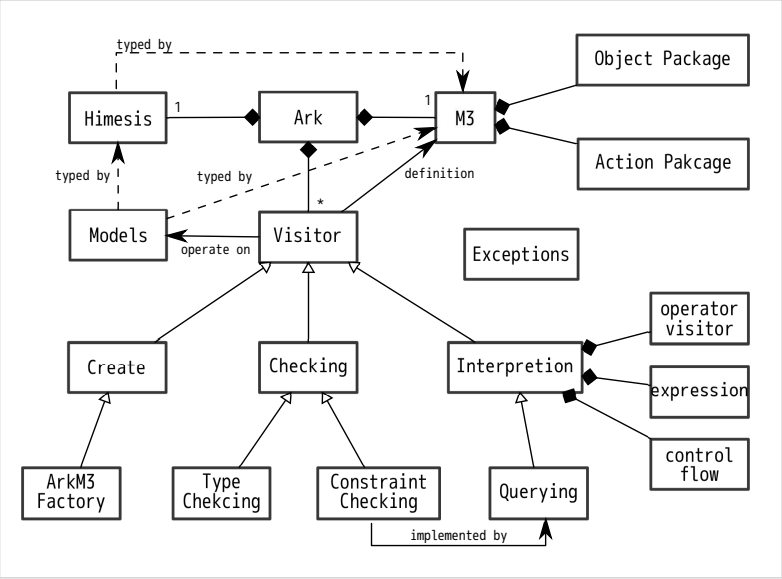


Figure 4.1: The Ark Composition

Ark (**A**ToMPM **r**eusable **k**ernel) is the minimal core of AToMPM—the metamodelling and transformation tool built on ArkM3—which provides fundamental support to metamodelling and transformation. Figure 4.1 presents the composition of Ark. It includes a self-described, executable meta-metamodel ArkM3 with which a tool can bootstraps, and a metamodel of HADT graphs that specifies models' internal representation. Ark conforms to the two-dimensional metamodelling architecture. It has ArkM3 in the logical dimension, while it adopts Himesis in the physical dimension. Ark has modules that achieve essential metamodelling functionality, including *Create*, *Checking* and *Interpretation*, and they can be further specified.

Ark is implemented in Python 2.5. Since Python is a convenient programming language for fast prototyping; and both Himesis and AToM^3 are implemented in Python, we held on to the same language. Python is an interpreted language so that the performance is not as fast, which was not AToMPM's primary concern. Although, at the end, the performance did turn out to be an important issue.

4.1 The Implementation of ArkM3

ArkM3 is predefined, so that the first-level classes are compiled to Python classes and stored in *.py* files. Whereas because ArkM3 is self-describable, it can be recreated from ArkM3 Python objects in the place of files and therefore be represented by graphs in memory. A graph model can then be output to and loaded from XML files using Himesis serialization tool.

4.2 The HADT Graph Himesis

AToMPM supports hierarchical, thus the HADT Graph Himesis is chosen instead of ASG.

Himesis was developed by Provost [Pro05] at MSDL. There are two packages in Himesis, one of which is for the hierarchical graph representation, and the other for graph utilities, such as graph matching and serialization. Ark adapts Himesis to meet the need its characteristic. We gave up Himesis' graph matching package because Himesis' matching is subgraph matching instead of pattern matching. It can only match graph structure (nodes and connections) but not the properties of the graph nodes which compromises Himesis' ability to be useful in model transformation. We temporarily keep the Himesis' serialization tool which exports models to and imports from XML and it uses folders to represent the insideness. More efficient exporter and importers would be helpful in the future.

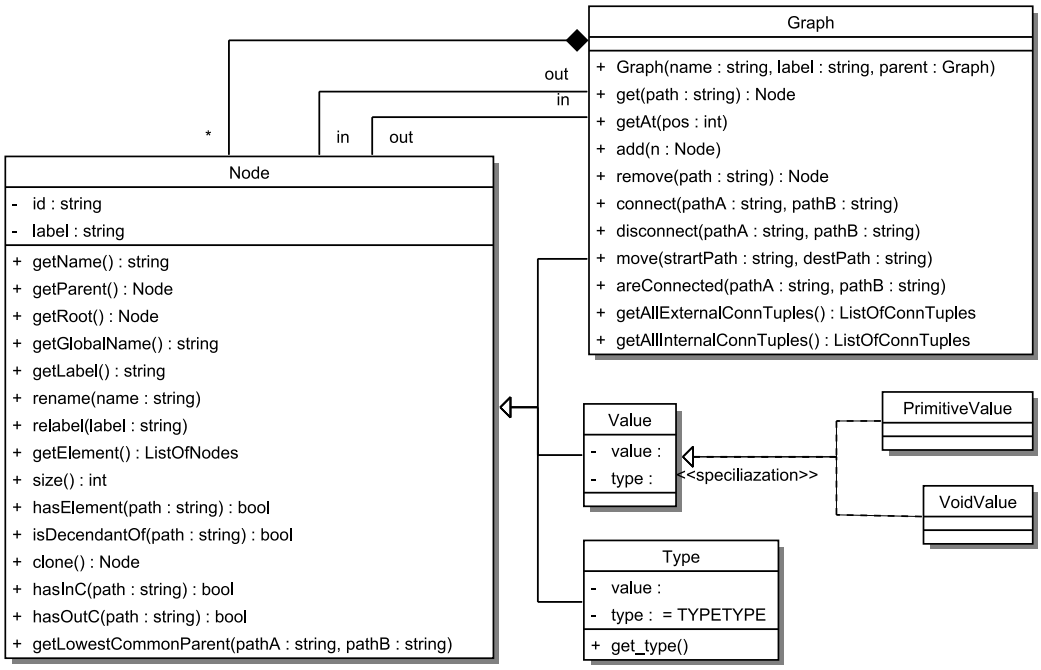


Figure 4.2: Modified Himesis Metamodel

The modified Himesis metamodel is shown in Figure 4.2. The constituent of Himesis models is the abstract class *Node*. Its name is defined by *id* and its type (metamodel) by *label*. *Graph* enable the hierarchy. Deriving from *Node*, *Graph* is the only object that can contain other graph. *Graph* also provides useful actions such as *add*, *remove*, *connect*, etc.

we modify Himesis to respond to ArkM3 Values and Types. Originally, Himesis provides four primitive types—Int, Float, String and Bool. The modified Himesis is in Figure 4.2. We define $Value$ and $Type$, $PrimitiveValue$ for primitives and $VoidValue$ representing any value that is undefined. These classes are all derived from Node but they can not contain other nodes. Besides *name* and *label*, they also have *value* and *type* (a string explicitly indicating type).

In addition, Himesis maintains the connection information globally in the root node for the sake of better performance.

Terminology. In the following sections, we will refer to the "modified Himesis" as "Himesis".

4.3 Mapping Between ArkM3 models and Himesis models

As discussed in Section 2.1.2, there is a projection between the logical dimension and the physical dimension in a two-dimensional architecture. As for AToMPM, every model in AToMPM is physically a graph, while at the same time, belongs to a logical domain. A strict mapping between graphs and logical models is important for AToMPM automatic generation, updating, interpretation and trace-ability.

Different meta-metamodels could have different mappings. It is because a meta-metamodel defines the syntax of a metamodel and the way to interpret and instantiate this metamodels. In this thesis, we only defined the mapping for ArkM3. If a modeller would like to define a new meta-metamodel, he has to define the specific mapping.

Mapping AToMPM logical models to Himesis is somehow similar to that of mapping from the abstract syntax of a language to the concrete syntax. Himesis models, due to their indifference towards the content, have to include the supplementary structure to indicate the role of a node. It shows the complete structure so that the system can parse and analyse. The other way around, the concept model in the logical dimension is smaller than the graph model because it only contains the human understandable objects and does not have extra structure to assist software interpretation. For mappings please refer to Appendix A. You may need to be aware of the mapping when reading the Chapter **Case Study**.

4.4 AToMPM Kernel Functionality

Ark demonstrates the use of the metamodelling architecture and the meta-metamodel with executability. As introduced in the beginning of this chapter, Ark uses visitors to realize functions of creation, update and deletion, conformance checking, interpretation, and serialization.

The Visitor Pattern

Ark uses Visitor Design Pattern to implement the functionality. Visitor Pattern is a way of separating an algorithm from an object structure upon which it operates. In essence, the Visitor family allows new methods to be added to existing constructs without modifying the constructs [Mar02]. Instead, one creates a visitor class that implements all of the appropriate specializations of the function [GHJV95].

The idea is to use a structure of classes which has an `accept` method that takes a `visitor` object as an argument as illustrated in Figure 4.3.

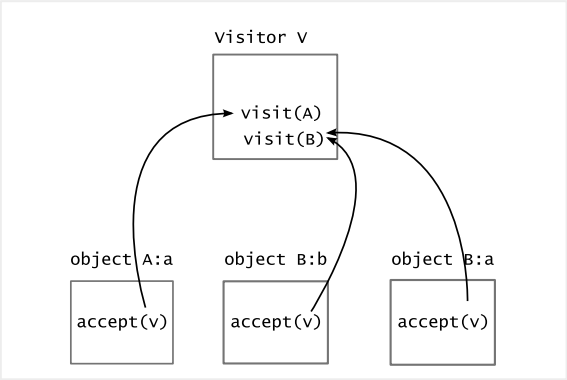


Figure 4.3: Objects Can Accept a Visitor

Nevertheless, because Python is a dynamic typing language, the visitor pattern is achieved using a `dispatch` method as the following.

```
def dispatch(self, node, args=None): # lookup node type
    self.node = node
```

```
        klass = node.getLabel()
        meth = self._cache.get(klass, None)
        if meth is None:
            className = klass.split(".")[-1]
            meth = getattr(self, "visit" + className, AttributeError)
        return meth(node, args)
```

The `self._cache` is to reduce the overhead of searching and the `meth` is a function pointer that redirect the execution to the right visitor function. Using the `dispatch` method is reasonable also because the models are projected to graph nodes, whose type are stored in the attribute *label* and can not be deduced by the compiler.

The visitors are aware of the metamodel. A visitor includes methods (as in Visitor `V`) for each class in the metamodel level. Its traverses the graph model, and whenever they reach a graph node the `dispatch` method redirects to the right visitor method according to the graph node *label*.

4.4.1 Search for a Model Element

Searching function is implemented via the Search visitor. As the models are internally represented in graphs, this visitor searches the model on the base of graph and nodes.

The search function is invoked by calling **search** method in the module `Ark.function.HmSearch`.

```
1  from Ark.function.HmSearch import HmSearch
2  HmSearch().search(root, path)
```

It takes two parameters, one of which is the root of a Himesis model and the other a string divided by '.' for the relative path, and the return value of this method is a graph node otherwise *None*.

```
1  def search(self, root, name):
2    self.namelist = name.split(".")
3    self._match = None
4    self.dispatch(root, None)
5      if self._match ==None:
6        print "Can not find "+ name +" in model "+ root.getLocalId()
7      return self._match
```

The **search** method firstly divide the path into items representing node local ids. The visitors traverse the model and at every level of containment, find the node whose name matches the current level item in path list. In line 4, we have **dispatch(Node elem, List args)** method redirects the execution to corresponding visitor method.

Figure 4.4 shows an example search.

Require: *elem* is *Himesis.Value* or *Himesis.Type*
1: **if** *elem* local Id == *pathList*[*depth*]: **then**
2: save the reference to *elem* in *self._match*
3: **return** *True*
4: **else**
5: **return** *False*
6: **end if**
Require: *elem* is *Himesis.Graph*
7: **if** *elem*'s local Id == *pathList*[*depth*] **then**
8: **if** *depth* refers to the last item in the *pathList* **then**
9: save the reference to *elem* in *self._match*

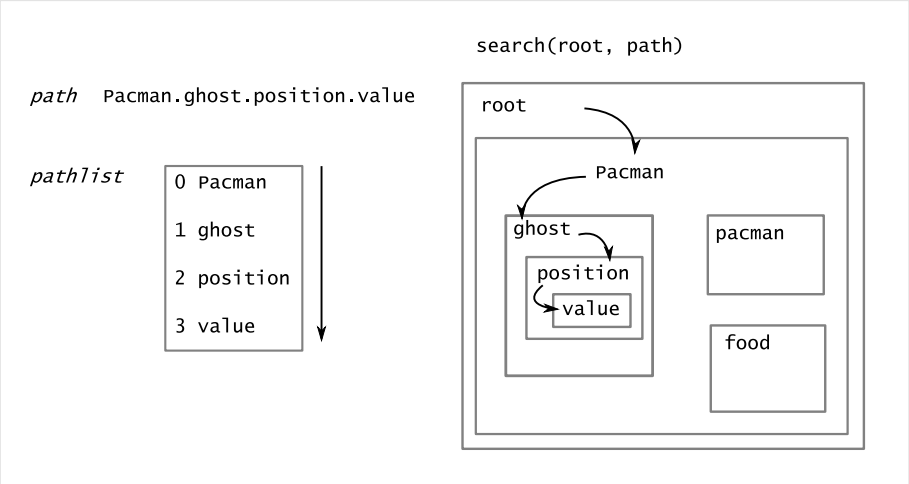


Figure 4.4: An Example of the Search Visitor

```
10:       return True
11:     else
12:       depth++ to match next level
13:       for all e in elem do
14:         if dispatch(e, args) is True then
15:           return True
16:         else
17:           depth– to match previous level
18:         end if
19:       end for
20:     end if
21: end if
22: return False
```

The path are divided by '.'. Each segment indicates searching is one step deeper inside the root. As the visitor traversing the model, each time it compares the segment and the node id, only when they are the same, the visitor traverses all the children and match with the next segment. A match is found only when entire path are same with that of the destination.

4.4.2 Create Models and Metamodels

Ark creates a model in accordance to the metamodel definitions. It applies to any metamodel and model pairs as long as the meta-metamodel is ArkM3. To create a model element, the modellers should first get the metamodel definition via the Search visitor. Then they can instantiate this metamodel definition by invoking the `create` method in *HmHmFactory* module in the function package.

```
1  from Ark.function.HmHmFactory import HmHmFactory
2  definition = HmSearch().search(metamodel, template_path)
3  HmHmFactory(root).create(definition, metamodel, args, parent)
```

We first search for the definition in the metamodel, and then call the **create** method. The inputs to this function are the `metamodel element`, the `metamodel`, `args` for list of value of model attributes, and `parent` node of the created model. The created model element is add to the model (is referred to

by `root`) automatically. Taking a Class instance as an example, the algorithm of `visitClass` method is shown below,

---

Require: Creating a instance of first-class object, parameter: $definition$, $args$, $parent$
1: create Himesis node t as the root of this object
2: add t to $parent$
3: $sups \Leftarrow$ all the super classes and general glasses of the definition
4: $props \Leftarrow$ all properties from $sups$
5: $compo \Leftarrow$ all composition from $sups$
6: **for all** m in $props$ **do**
7: examine the $cardinality$, $uniqueness$ and $order$
8: create proper $DataValue$ to the above attributes
9: **repeat**
10: **dispatch**: definition m, root t, arguments
11: **until** meet the cardinality requirement
12: **end for**
13: **for all** m in $compo$ **do**
14: examine the $cardinality$, $uniqueness$ and $order$
15: create proper $DataValue$ to the above attributes
16: **repeat**
17: **dispatch**: definition m, root t, arguments
18: **until** meet the cardinality requirement
19: **end for**
20: $assoc \Leftarrow$ all named association ends from $sups$
21: **for all** a in $assoc$ **do**
22: examine the $cardinality$, $uniqueness$ and $order$
23: create proper $DataValue$ to the above attributes
24: **repeat**
25: add to t node named by the association end
26: **until** meet the cardinality requirement
27: **end for**

Creator first obtains all the meta information (properties, compositions, associations), including those in inherited and specialized classes between line 3 and line 5, and in line 20. Then the creator goes through all them and create Himesis graph. The created Class instance is added to the parent node if it is given (line 2). Finally, the element is returned. There can not be two elements with same id in one graph node. Therefore, in the creation visitor, the user should specify the id of an element, otherwise the visitor name the object with a unique number.

For faster generation of ArkM3 models, we also provide a factory module named `ModelCreator` for direct creation ArkM3 instances.

4.4.3 Conformance Checking and Constraint Checking

Model checking is used to check the conformance between model and the corresponding metamodel (metamodels, if applicable). The input shall be the model and its metamodel(s). By the content of checking and the way checking is performed, model checking has two parts. Type conformance checking verifies if the model conforms to the properties and relations defined in the metamodel. Whereas constraint checking checks if the model is consistent with the constraints defined in metamodel(s).

Type Conformance Checking

Type conformance checking is to check model elements according to the attributes, the compositions and the associations that are defined in the metamodel. The checking function is invoked by calling **check** method in the `HmTypeCheck` module, and returns the list of error messages encountered in the

process of checking.

```
1  from function.ArkCheck import ArkCheck
2  msg = ArkCheck().check(model, definition, metamodel)
3  print msg
```

The code of **check** is as follows,

```
1  def check(self, model, definition="", metamodel="", wt=False) :
2    self.metamodel = self.validateinput(metamodel, self.metamodel)
3      self.m = self.validateinput(model, self.currentdir)
4      self.mm = self.validateinput(definition, self.workspace)
5      if self.m == None or self.mm==None:
6          return self.msg
7    if !self.matchmodels(self.m, self.mm):
8        self.msg.append(["Metamodel of",name_to_match,"in",metamodel_name,"does not exist!"])
9    else:
10       self.dispatch(self.mm, self.m)
11       self.checkAllAssociations(self.workspace, self.m)
12   return self.msg
```

In line 2, it validates the inputs in order to make sure that the model, its definition and the metamodel actually exist and they are proper Himesis graph models. Then it examines if the model is actually typed by the metamodel in line 7. In line 10, **dispatch** redirects to the visitor that inspects the properties of model. Then **checkAllAssociation** verifies the connections and multiplicity between first-class objects.

```
1: model['isAbstract'] should be false
2: mcontent ⇐ components in model
3: sups ⇐ all the super classes and general glasses of the definition
4: props ⇐ all properties from sups
5: compo ⇐ all composition from sups
6: assoc ⇐ all named association ends from sups
7: for all m in mcontent do
8:   if m's definition in props then
9:     checkpair_p ⇐ (props[m.getLocalId()],m)
10:  else if m's definition in compo then
11:    checkpair_c ⇐ (compo[m.getLocalId()], m)
12:  else if m's definition in assoc then
13:    checkpair_a ⇐ (assoc[m.getLocalId()), m)
14:  else
15:    print self.msg.append(["ModelError", m, "is not defined."])
16:  end if
17: end for
18: if anything left in props then
19:   print self.msg.append(["ModelWarning", m, "is not instanticated."])
20: end if
21: for all elem in checkpair_p, checkpair_c, checkpair_a do
22:   self.dispatch(elem[0], elem[1])
23: end for
```

The checking visitor traverses the model packages in the depth-first order. When it finds a first-class object, such as instances of Class, Association and Composition, The **dispatch** method directs the exe-

cution to the corresponding visitor. Taking the Class instance as an example, similar to that in creation, when a **visitClass** method is called, the visitor gets the actual content in the checked model (Line 2) retrieves the corresponding metamodel element(s). In line 4, 5 and 6, we collect all the meta information from metamodel, and each line returns a list. Between line 7 and line 17, the visitor compares these lists and the model content list for mismatches. At last, the visitor checks each model and metamodel pair to find inconsistency as in line 22.

```
1: associations ⇐ all the instances of Association
2: for all a in associations do
3:    definition ⇐ metamodel of the object that a connect to
4:    mfrom ⇐ incoming link of definition['from']
5:    from ⇐ incoming link of a['from']
6:    if mfrom's id != from's type then
7:       print Report Connection Error
8:    end if
9:    mto ⇐ incoming link of definition['to']
10:   to ⇐ incoming link of a['to']
11:   if mto's id != to's type then
12:      print Report Connection Error
13:   end if
14: end for
```

The **checkAllAssociations** gets all the instances of Association of the model. For each of them, it finds the definition in the metamodel. It examines the designated names that *to* and *from* are connected to in line 4 and line 5, and examines the type of the actually connected object in line 6. If the name and the type are not match, an error message is created.

Constraint Check

The constraint checking checks if a model is consistent with the constraints by interpreting the constraint expressions attached to the model elements. The constraints can locate either in the metamodel or in the model. It is invoked by the **check** method in the `ConstraintCheck` visitor.

```
1
2  from function.ConstraintCheck import ConstraintCheck
3  msg = ConstraintCheck().check(model, definition, metamodel)
4  print msg
```

The constraint checking visitor traverses the model. For each model object, it firstly examines the constraints of the object. Then the visitor searches for the metamodel definition and all related metamodel elements classes) and exams the constraints defined for the model. Evaluating the constraints is implemented by interpreting constraint models (as action models being interpreted in Section 4.4.4). It returns either `True` or `False`.

```
1
2  from function.HmActionInterpreter import HmActionInterpreter
3  if HmActionInterpreter().execute(expr, model, metamodel) == false:
4     self.msg.append(["ConsraintError", expr,"in", model, "and", metamodel])
```

4.4.4 Interpretation

Executability is an important feature of Ark. Ark executes the action models so as to simulate, transform models. Execution is also used when the status of models are queried.

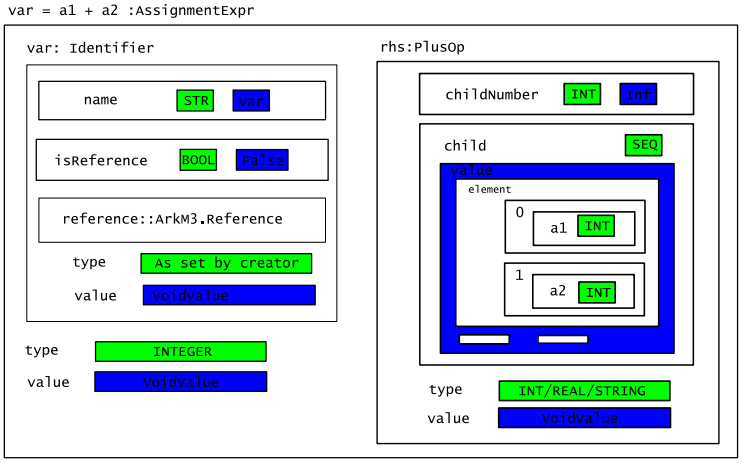


Figure 4.5: An Action Model Example Describing *var = a1 + a2*.

The program is stored in a structure such as that in Figure 4.5. Because each graph node can have at most one parent, therefore the action model is actually a tree. AToMPM executes action models is by traversing and interpreting. An interpretation visitor is used to interpret the action models. To invoke the interpretation, you can use the following lines,

```
from function.HmActionInterpreter import HmActionInterpreter
HmActionInterpreter().execute(actmodel, hostmodel, metamodel, parameters)
```

The Interpreter Visitor is set up as such,

```
1 class HmActionInterpreter(Visitor):
2   def __init__(self):
3     self.model = None
4     self.metamodel = None
5     self.resultStack = []
6     self.context = list()
7     self.history = dict()
```

The variables represent respectively: line 1, the host model of the constraint; line 2, the metamodel of the action's host model, added to enable the create function; line 3, a stack which stores the result of evaluated expressions; line 4, the context of the executing, the root of constraint model get executed; line 5, the stack for function call invoked. The system traverse the action model tree pre-orderedly. The evaluation of expressions are pushed in the result stack, whose function is same with the runtime stack in a compiler. The remained value in the stack after the execution is the return value of the action model.

```
1   def execute(self, actmodel, hostmodel, metamodel, parameters)
2         # record the curent activated action model
3         self.context.append(act)
4
5         self.metamodel
6         self.actmodel
7         # clear the stack, meaning the previous result stack
8         # need to be stored once a execute function is called.
9         self.resultStack = []
```

```
10
11      # begin to execute the action model
12          self.dispatch(act, parameter)
13
14          # return the result and clean the stack
15          if len(self.resultStack) == 1:
16              result = self.resultStack[0]
17          else:
18            result =  self.resultStack
19
20          # remove the current action model from the context list before returning the result.
21          self.context.pop()
22
23          return result
```

execute method parameter include the action model, the hosting model and the metamodel of the hosting model, as well as the parameters passes to the action model if any.

4.4.5 Transformation and Simulation

Transformation is not directly supported in AToMPM, as the pattern matching is not supported in Ark. It belongs to AToMPM extension for transformation. The transformation rule actions model can be built with these primitives. These actions are executable in the AToMPM interpreter. For a description of the execution of action models refer to Section4.4.4. For the example of model transformation please refer to the Case Study in the next chapter 5.

The case of simulation is similar. Please refer to the Case Study for the example.

In this chapter, we explained the implementation of the kernel of the metamodelling tool AToMPM. The following chapter will present an case study, a Reader/Writer System described in Petri Net graph, to illustrate the use of this kernel.

5

Case Study: Readers/Writers System in Petri Net

In this chapter, we present a Readers/Writers System Petri Net model to demonstrate the use of the AToMPM kernel. This example shows the AToMPM approach to define metamodels, instantiate metamodels, verify models and analyse models, as well as how the models are stored and retrieved in this metamodelling system.

Chapter Organization. In Section 5.1, we introduce the Readers/Writers System and Petri Net formalism. In Section 5.2, We show ArkM3's approach to create metamodels of two different kinds of Petri Net formalism, which are Place/Transition Petri Net (PTPN) and Capacity Constraint Petri Net (CCPN). In Section 5.1.2, we create an Constraint Capacity Petri Net model by instantiating CCPN formalism. In Section 5.4, we check the conformance between the CCPN model and the CCPN metamodel, transform this model to a equivalent PTPN model, and simulate the resulted model by executing transformation rules in the AToMPM interpreter.

5.1 Background

5.1.1 Readers/Writers System

We reuses the Readers/Writers System example from [MBC+94] to explain the use of Ark. A Readers/Writers System includes a set of processes that may access a common database either for retrieving or updating. Any number of readers may access the database concurrently. A writer, however, requires exclusive access to the resource. If one process is writing to the database, no other processes, including readers, may have access to the database. There are three interesting aspects of this system making it amenable to the Petri Net modelling.

- concurrency: two processes may be concurrently accessing the database for reading;
- choices: an access can either be a read or a write;
- mutual exclusion: only one process at a time may access the database for writing.

5.1.2 Petri Nets Formalism

The Petri Net was first developed by C. A. Petri in the early 1960s. It provides a set of graphical notation and operational semantics well suited for modelling the concurrent systems as well as the discrete event systems. A Petri net is a graph defined as such in [CL08],

$$(\boldsymbol{P}, \boldsymbol{T}, \boldsymbol{A}, \boldsymbol{w})$$

- $P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of places;
- $T = \{t_1, t_2, \ldots, t_m\}$ is a finite set of transitions;
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of arcs from places to transitions and from transitions to places;
- $w : A \rightarrow \mathbb{N}$ is a weight function on the arcs.

A Petri net consists of places (are usually represented by circles) and transitions (are usually represented by bars) which are connected by directed arcs. An arc connects a place to a transition or vice versa, but never between places or between transitions. The places connected by arcs to a transition t_j are called the input places $I(t_j)$ of that transition and the arcs are called input arcs; the places connected by arcs from a transition t_j are called the output places $O(t_j)$ of that transition and the arcs are output arcs. Each arc is accompanied by a weight representing the enabling condition.

A place may contain any non-negative number of tokens. When the number of the tokens is assigned to the places, the places are marked and the Petri Net become a marked Petri Net. A state of a marked Petri net, which is sometimes called marking, is the number of tokens in all the places, as such,

$$\mathbf{x} = [\boldsymbol{x(p_1), x(p_2), \ldots, , x(p_n)}]$$

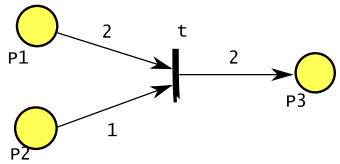


Figure 5.1: Example of the Petri Net Enabling Rule and Firing Rule

The state transition mechanism is captured by the structure of the Petri Net graph. The operational semantics of the Petri Net is related to the "enabling rules" and the "firing rules". These rules are associated with transitions. The enabling rules state the conditions under which transitions are allowed to fire, while the firing rules define the marking modification induced by firing the transition. A transition t is enabled if and only if each input place $I(t)$ contains a number of tokens greater than or equal to a given threshold $w(p,t)$, the weight on that arc. Taking the Petri net in Figure 5.1 for example, t is enabled if the number of tokens in the place $P1$ is not less than 2, and the number of tokens in the place $P2$ is no less than 1.

Petri Net states are changed by firing of the enabled transitions. When a transition t fires, it consumes from each place p_i in its input places set $I(t)$ as many tokens as the weight $w(p_i,t)$ on the arc connecting p_i to t, and puts to each place p'_i in its output places set $O(t)$ as many tokens as the weight $w(t,p'_i)$ of the arc connecting the t to p'_i.

$$\boldsymbol{x'(p_i) = x(p_i) + w(p_i,t_j) - w(t_j,p_i)}$$

In the Figure 5.1, when t is fired, 2 tokens will be removed from $P1$ and 1 from $P2$, and then 2 tokens are added to $P3$. A Petri net repeatedly fires the enabled transitions until there are no more.

Among the variances of the Petri Net formalisms, we choose the Place/Transition Petri Net (PTPN) and the Capacity Constrained Petri Net (CCPN). They are both simple to understand and complex enough to demonstrate ArkM3's important features.

CCPN is different with PTPN in that CCPN places have an extra attribute *maxToken* indicating the maximum number of the tokens in a place as well as a corresponding constraint to ensure the number of tokens is smaller than *maxToken*. Whereas, PTPN nets express the same meaning in a different manner, that is to add a reverse link in CCPN graph structure. Figure 5.2 shows a pair of equivalent CCPN net and the PTPN net implying P may have 0 to 3 tokens.

In the CCPN net on the left, $max = 3$ indicates if P has 3 tokens at the most. Whereas, in the PTPN net on the right, in place of the attribute max, we have a reversed link between t_2 and t_1. This link consists of one place p' with 3 tokens and two arcs with the same weight to their opposite counterparts. When the number of token in p is smaller than 3, every time when t_1 is activated, 1 token is sent to p and at the same time, one removed from p'. When there are already 3 tokens in p, the number of tokens in p' is 0. Thus, $t1$ is deactivated until tokens are removed from p. We are going to demonstrate such CCPN to PTPN transformation in the following sections.

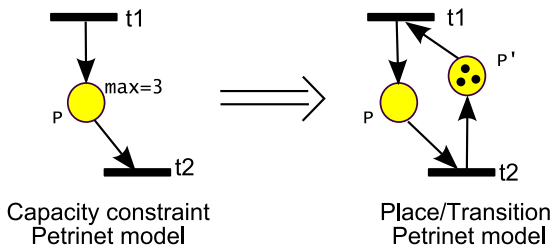


Figure 5.2: Equivalent CCPN and PTPN Models

5.2 Create Petri Net Metamodels

From this section, we start our metamodelling process. The first step is to create the metamodels for both formalisms.

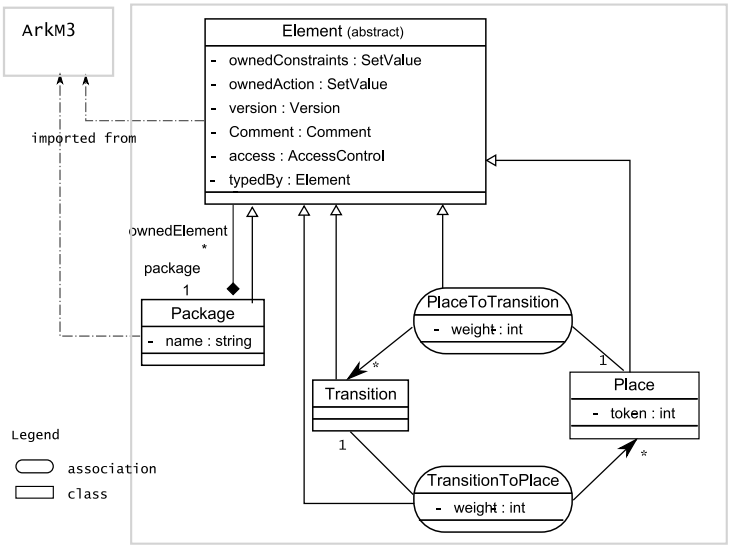


Figure 5.3: Place/Transition Petri Net Metamodel

Figure 5.3 shows the ArkM3 class diagram for the PTPN metamodel. Besides the Petri Net's *Place*, *Transition*, input arc *Place2Transition* and output arc *Transition2Place*, this metamodel also imports two classes, *Element* and *Package*, from ArkM3. This importation, on the one hand, ensures the strict metamodelling hierarchy—the two classes now belong to the PTPN metamodel rather than ArkM3, they are instances of ArkM3's *Class* and they are respectively inherited in PTPN metamodel and instantiated in PTPN models. On the other hand, this approach includes ArkM3's expressiveness in the metamodel. By importing the class *Element* and inheriting from it, we not only import its attributes but also its associations, so that we are able to describe useful informations of an PTPN metamodel element, from model version to the constraints and the actions. By importing *Package* and instantiate it in the next level, we have a container of PN models.

Comparing with PTPN, CCPN has an extra attribute indicating the maximum number of tokens in a place. Correspondingly, *Place* has a constraint that the number of tokens in a place is less than *maxToken*. Figure 5.4 shows the CCPN metamodel class diagram and highlights the difference.

To build such a metamodel in the kernel, the following steps are to be followed. You may need to refer to Section 4.3 to understand the path of some elements. First of all, we load the meta-metamodel into

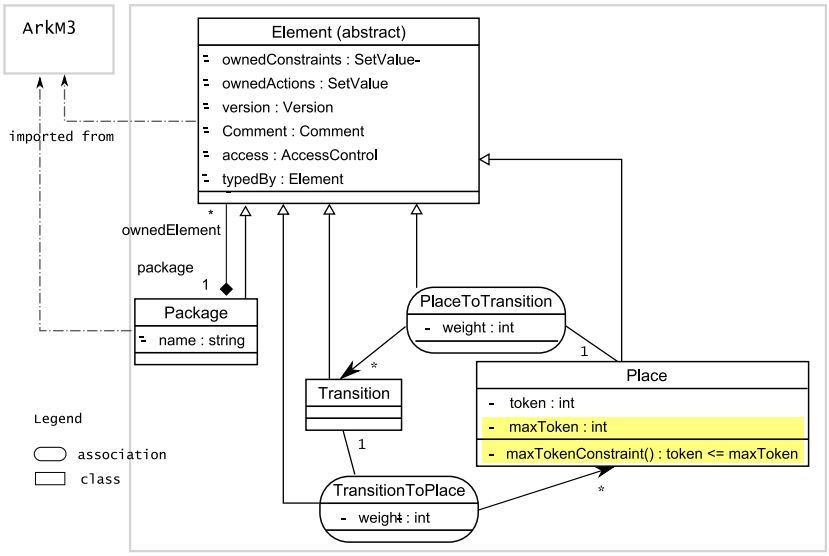


Figure 5.4: Capacity Constrained Petri Net Metamodel

the metaverse and set up the environment:

```
workspace = METAVERSE
```

creates a local reference to METAVERSE which represent a subsection of it. It can be a subsection of the metaverse, or a subsection of some workspace.

```
m3 = createArkM3_in_Hm()
```

loads ArkM3 into the memory.

```
workspace.add(m3)
```

adds loaded ArkM3 to the metaverse.

```
HashmapGenerator().createHashMap(METAVERSE, localroot=m3, metaverseid = "AToMPM_Workspace")
```

initialises the AToMPM hash table, loading frequently used ArkM3 components in to hash table.

We take the creation of the CCPN class Place as an example to show how the metamodel elements are created and how they are connected using Python code.

1. Create the instance of *Package* to hold all the Petri Net elements and add it to the current *workspace* in the $METAVERSE$.

   ```
   metamodel = ModelCreator().createPackage("PetriNet_META_Hm", None)
   workspace.add(metamodel)
   ```

2. Create an abstract class `Element` as the super class of all the classes in this metamodel. This is considered as importation from ArkM3. The same is for concrete class `Package` who inherits from `Element`. `root` indicates the parent Himesis node of the created element.

   ```
   superClass = factory.createClass("Element", root = metamodel["ownedElement.value.element"],
   isAbstract=True, package=metamodel)
   ```

3. `Place` instantiates the metamodel Class and is added to the package referred by the variable `metamodel` in Line 1. `Place` inherits from `Element` in Line 2 and 3. Increase the element counter of the package by 1 in the last line. `Transition` is defined in the same way.

```
place = f.createClass('CCPetriNet.Place', package=metamodel)
place.add_super_class(superClass)
metamodel.connect(place["super.value.element.1"], superClass)
metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].
getValue()+1)
```

4. Create the property `token` which is a property owned by `Place` (reside in the Himesis node `place["ownedProperty.value.element"]`). Its default value is 0. There is one such property in each `Place`. Similar process is to the property `maxToken`.

```
factory.createProperty("token", root=place["ownedProperty.value.element"], typ="INTEGER",
isComposite=True, lower=1, upper=1, default = 0,hostClass = place)
```

5. Define an association `Place2Transition` Line 1 connecting `Place` to `Transition` and its cardinality Line 2 and 3, with a many to many relation. The same process for `Transition2Place`.

```
pl2tr = factory.createAssociation("Place2Transition", root=metamodel["ownedElement.value.element"],
package=metamodel, isFrom = place, isTo = trans)
factory.createProperty("inCardinality", root = pl2tr, isComposite=True, isUnique=True,
isOrdered=False, lower=1, upper=Inf(), hostClass=pl2tr)
factory.createProperty("outCardinality", root = pl2tr, isComposite=True, isUnique=True,
isOrdered=False, lower=1, upper=Inf(), hostClass=pl2tr)
```

We also need to insert the constraint to the Place.

```
1 Constraint maximumTokenConstraint:
2   self.token <= self.maxToken
```

The following code implements the `maximumTokenConstraint`. Line 1 instantiates the Constraint class, which is associated with `Place`. Then we build the model for the expression from left to right. The left hand identifier `t` refers to the "token" of the resided class while the right hand side identifier `mt` refers to "maximumToken". The last line create an instance of `no greater than` operator, whose two operand is `t` and `mt`.

```
const1 = factory.createConstraint("maximumTokenConstraint", root=place["ownedConstraint.value.element"],
condition="CONDITION", isImplemented = True, hostElement = [place])
lhs = factory.createIdentifier("t", isRef = True, reference = factory.createReference(id="ref",
ref=["SELF","token"], meta="ArkM3.AL.IdentifierReference"))
rhs = factory.createIdentifier("mt", isRef = True, reference = factory.createReference(id="ref",
ref=["SELF","maximumToken"], meta="ArkM3.AL.IdentifierReference"))
factory.createNotGreaterThan(root = const1["expression"], child = [lhs,rhs])
```

For the complete CCPN metamodel code please refer to Appendix B.

Note that Element and Package are needed in all levels in the metamodelling hierarchy. Element is the root of all the elements in a model. All classes are subclass of Element as allows model objects to have constraints, actions, access control and version control. Package class indicates the elements belonging to the same model and to be handled together. Element and Package is imported to the next level during metamodelling, so that we can have their features without violating the strict metamodelling principle.

5.3 Create a Petri Net Model

Now we can create the Petri Net model. In Figure 5.5, we design the graphical representation of a CCPN model for the Readers/Writers System.

It comprises nine places $P = \{p1, p2, \ldots, p9\}$ and seven transitions $T = \{t1, t2, \ldots, t7\}$. Transition $t1$ is connected to $p1$ through an input arc, and to $p2$ through an output arc. Place $p5$ is both input and output for transition $t4$. According to the definition of parametric initial marking, this initial situation can be expressed with the vector $x = \{1, 0, 0, 0, 0, 1, 0, 0, 0, 0\}$. $p0$ pertains one token so as to provide

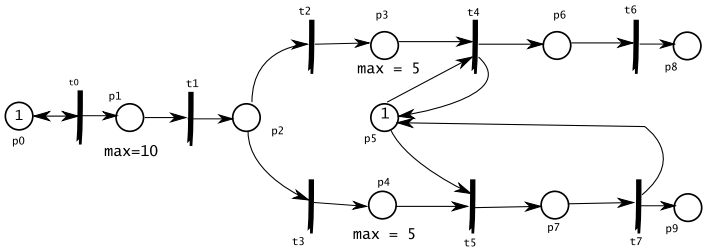


Figure 5.5: The CCPN Model of a Simple Readers/Writers System

infinite service request. $p5$ contains one token. It controls both route $t4, p6, t6$ and $t5, p7, t7$. It indicating the system can accept multiple reading request but can only process on writing request at one time.

Similar to that of creating a metamodel, the process include searching for metamodel, instantiation and connecting the models.

1. Use `Factory` visitor and `Search` visitor.

```
factory  = HmHmFactory(root=workspace)
searcher = HmSearch()
```

2. Initialise the `Search` visitor for CCPN metamodel.

```
mm = searcher.search(root, directory)
```

3. Taking "CCPetriNet_META_Hm.Place" as an example, we search for the metamodel and the class definitions in the `workspace`. The same procedure applies to Package, Transition, and the associations.

```
mm = searcher.search(root, directory)
place_template = searcher.search(mm, "CCPetriNet_META_Hm.Place")
```

4. We instantiate the metamodel definition using `factory.create`, arrange its inheritance and specialization, add it to the package and increase the element counter. add properties to the class. and define constraints and actions if any.

```
place1 = factory.create(place_template, mm, [pack["ownedElement.value.element"], ID1, None])
```

5. Assign value to the class property

```
place1["token.value"].setValue(8)
```

6. Connect the `from` of this association with the `end` classes represented by `ID1` and `ID5`.

```
pack["ownedElement.value.element"].connect( IDp2t2+".from", ID1)
pack["ownedElement.value.element"].connect(IDp2t2+".to", ID5)
```

5.4 Use Ark Functionality

5.4.1 Model Checking

Having all the required models, we can do analysis and simulation in AToMPM.

First we want to know if our Petri Net model in the previous section is legal. We can use model checking function. Model checking is to check the consistency between model and corresponding metamodel. To check consistency between a Petri Net model with CCPN metamodel, we shall use the following command:

```
#Type check model against CCPN metamodel"
from function.HmHmTypeCheck import HmHmTypeCheck
print HmHmTypeCheck().check(model, meta)

# Type check model against CCPN metamodel"
from function.HmHmConstraintCheck import HmHmConstraintCheck
print HmHmConstraintCheck().check(model, meta)
```

The result of conformance checking is a list of all the errors and warnings found in the model, which here are displayed in lines in the console.

5.4.2 Transform the CCPN model to PTPN model

Similar to that in AToM^3 transformation, we defines rules consisted of conditions (LHS and RHS) and transformation actions for model transformation. The rules have priority the only when the rules of higher priority do not have any more matches, the next highest rule can fire. Executing transformation rule involves two steps, first of which is to search for the patterns in the graph and the second is apply the transformation action to the result of first step.

Note that because transformation is planned to be tended by a special formalism in AToMPM, the kernel does not design special API for the transformation. So far, hard coded program based on Himesis graph search is used for pattern matching; the rule actions are modelled by ArkM3 action model and are executed by interpretation visitor.

As introduced in the previous Section 5.1.2, the main difference is between including a attribute and having it restricted be structure. So we make this difference the first transformation rule. It identifies the critical structure to change. This rule is of top priority.

```
def search_CCPN2PN_Pattern_A(model):
    """
    print " CCPN2PNpattern   ==     CCPetriNet_META_HmTransition.:b"
    print "                   \      CCPetriNet_META_Hm.Transition2Place:c"
    print "                   O       CCPetriNet_META_Hm.Place:a"
    print "                   /      CCPetriNet_META_Hm.Place2Transition:d"
    print "                  ==     CCPetriNet_META_Hm.Transition:e"
    """
```

After that it comes to the following rules to transform CCPN elements to PTPN elements. The rules including the conditions and actions are list below. As in following code, the LHS and negative condition is the pattern to be searched in the graph. Searching for pattern is written in Python as,

```
for i in model["ownedElement.value.element"].getElements(labels=["CCPetriNet_META_Hm.Transition2Place",]):
    if not i["to"].getExternalConnTuples(): continue
    for j in model["ownedElement.value.element"].getElements(labels=
    ["CCPetriNet_META_Hm.Place2Transition",]):

        if not j["from"].getExternalConnTuples():
            continue

        if i["to"].getExternalConnTuples()[0][1].getGlobalId()
        == j["from"].getExternalConnTuples()[0][1].getGlobalId()
        and i["from"].getExternalConnTuples()[0][1].getLabel() == "CCPetriNet_META_Hm.Transition"
        and j["to"].getExternalConnTuples()[0][1].getLabel() == "CCPetriNet_META_Hm.Transition"
        and i["to"].getExternalConnTuples()[0][1].hasElement("maximumToken"):
            # If the place is not already converted, add it to the return value.
            tag = True
            for o in i["to"].getExternalConnTuples()[0][1].getExternalConnTuples():
                if o[1].getLabel().split(".")[0]  == "PetriNet_META_Hm":
```

```
                tag = False
                break
```

The result of pattern searching is returned as,

```
c = i
d = j
b = i["from"].getExternalConnTuples()[0][1]
e = j["to"].getExternalConnTuples()[0][1]
a = i["to"].getExternalConnTuples()[0][1]
return a, b, c, d, e
```

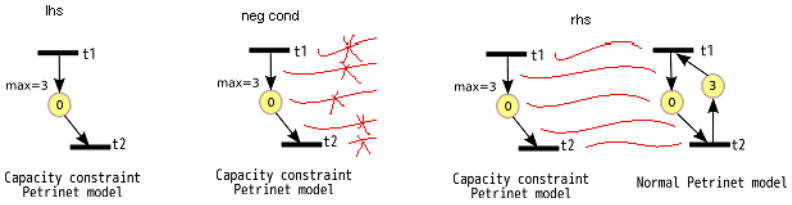


Figure 5.6: CCPN to PTPN Model Transformation Rule 1

The next set of transformation rules is to transform CCPN model elements to corresponding PTPN model elements. As you can see in Figure 5.7, Generic Link is the link to assist the transformation pattern searching. It is used to mark the correspondence of models in the source and the destination.

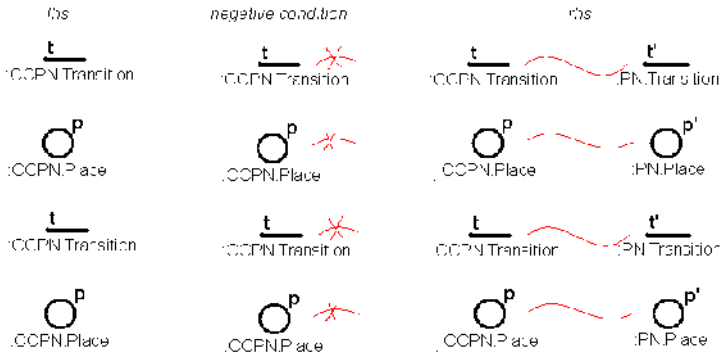


Figure 5.7: CCPN to PTPN model transformation Rule from 2 to 5

The last part is to delete the generic links and the original CCPN model elements. This rule is of least priority, that it is fired only when no match can be found for rule 1 - 5. This is described as rule 6–9.

The transformation rules, including the pattern searching and rule action models are pre-defined in python files named search_CCPN_Pattern_X and rule_X_actioncode_model.

Let's take the first rule as an example to show how it is implemented.

```
from searchCCPN2PNPattern import search_CCPN2PN_Pattern_A
[a, b, c, d, e] = search_CCPN2PN_Pattern_A(model)
```

The above code call for a pre-defined, hand-coded search function that describe the graph structure on the left-hand-side. Then it returns a list of the CCPN nodes that match the pattern. In the following code, we load the pre-defined action models from **rule_A_actionHm_model**, which takes the search

result as input. We send this action model from the last step to the HmActionInterpreter. By applying **execute**, this part the CCPN model is converted to PTPN.

```
from Example.ccpn2pn_hm_rule import rule_A_actionHm_model
from function.HmActionInterpreter import HmActionInterpreter
while a and b and c and d and e:
    y = rule_A_actionHm_model(model, [a, b, e, d, c], meta2)
    HmActionInterpreter().execute(y, model, meta)
    [a, b, c, d, e] = search_CCPN2PN_Pattern_A(model)
```

Iteratively search and execute the transformation rules, the transformation from CCPN model to PTPN model is implemented.

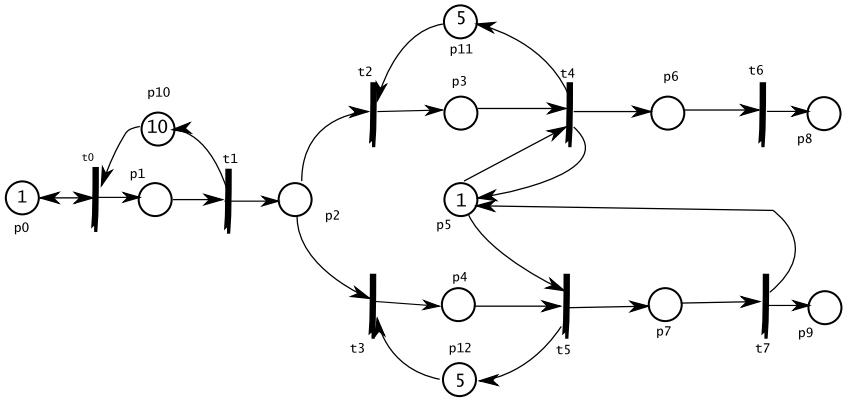


Figure 5.8: Transformation Result is a PTPN Model of the Readers/Writers System

After the transformation, the model in Figure 5.5 is changed to the one in Figure 5.8.

5.4.3 Simulate PTPN Dynamics

This section shows simulating the Readers/Writers model by the means of model transformation. The simulation follows the Petri Net enabling and firing rules mentioned in Section 5.1.2.

Simulating a Petri Net model, slightly different from the transformation between CCPN to PTPN, the input of the simulation is one metamodel (PTPN) along with one model (instance of PTPN). There is only one rule needed which searches for enabled transitions and randomly fire one. The simulator scans the whole model and returns the sets of enabled transition and the related model objects. Then, the system randomly select one set and hands it over to firing, the simulator will then apply the action model to this set of nodes and change their status.

To search for the enabled transition and its input places and output places, we can use some code (pseudo code) as following.

```
def search_PN_Pattern_A(model):
    """
    print " PNpattern        ==     PetriNetTransition.:a"
    print "                   \      PetriNet.Transition2Place:b"
    print "                   0       PetriNet.Place:c"
    print " All the inputs to a need to satisfy place.token>p2t.weight"
    """
    from ArkM3.visitor.Factory import Factory
    f = Factory()
    a =[]
    c=[]
    for i in model['ownedElement'].getElements(labels=['PetriNet.Transition',]):
```

```
        tag = True
        if i.getExternalConnTuples():
                    for n in model['ownedElement'].getElements(labels=['PetriNet.Place2Transition',]):
                        if n['isTo'].getExternalConnTuples()[0][1] is i:
                            if n['isFrom'].getExternalConnTuples()[0][1] and n['isFrom']\
                            .getExternalConnTuples()[0][1].getLabel() == 'PetriNet.Place':
                                    t1 = n['isFrom'].getExternalConnTuples()[0][1]['token.value']\
                                    .getValue().value
                                    t2 = n['weight.value'].getValue().value
                                    if t1 <t2:
                                        tag = False
                                        break
                                    else:
                                         tg2 = True
                                         for o in c:
                                             if o.get(f.createIntegerValue(0)) is \
                                             n['isFrom'].getExternalConnTuples()[0][1] and \
                                             o.get(f.createIntegerValue(1)) is n:
                                                 tg2 = False
                                                 break
                                         if tg2 == True:
                                             c.append(f.createTupleValue([n['isFrom']\
                                             .getExternalConnTuples()[0][1], n]))
                    if tag == False:
                        a =[]
                        c =[]
                        continue
                    for m in model['ownedElement'].getElements(labels=['PetriNet.Transition2Place',]):
                        if m['isFrom'].getExternalConnTuples()[0][1] is i and\
                        m['isTo'].getExternalConnTuples()[0][1]  and\
                        m['isTo'].getExternalConnTuples()[0][1].getLabel() == 'PetriNet.Place':
                            a.append(f.createTupleValue([m['isTo'].getExternalConnTuples()[0][1], m]))
                    break
    return a,c
```

a and c represent the list of input places and output places respectively. They are the input to the firing rule, which substracts tokens from each of the input set (using a **mapping** operator) and adds to the output set. The exact action model please refer to Appendix C.

Summary

In this chapter, we presented a case study of the Readers/Writers system using Petri Net formalism. This case study demonstrated how to use AToMPM interfaces to build metamodels, instantiate metamodels, and check model conformance. It also showed the Ark's temporary approach for model transformation.

6 Conclusion

In this thesis, we adapted a two-dimensional metamodelling architecture that separates the view of modellers and that of tool developers. We presented the design of a general-purpose, self-describable, executable meta-metamodel ArkM3. With both, we make a general-purpose, comprehensive, bootstrapped metamodelling tool possible. We implemented the AToMPM kernel and presented the case study to demonstrate the benefit of our design.

We discussed the importance of clearly separating the physical dimension and the logical dimension, which allows tool developer and modellers work from on their own points of view without being distracted by the responsibility of the other. Bearing this in mind, we proposed a two-dimensional metamodelling architecture. We adopted the layered architecture for both dimensions. At the root of the logical dimension is ArkM3, while that of the physical dimension is Himesis, the metamodel for hierarchical, typed, attributed, directed graphs.

We have explained why including executability in the meta-metamodelling level is useful. We designed ArkM3 for such an meta-metamodel with executablility. ArkM3 is built on the basis of OMG's EMOF for its simplicity and popularity. To allow least dependence over the implementation, we also add Types and Values. Then we introduce an Action Language to ArkM3 by carefully choosing the control structure and a minimal set of primitive operators according to the need for the modelling and transformation.

We also introduce a hierarchical model management system, so called the "*Metaverse*". The metaverse contains all the models that are created, creating and to be created. AToMPM workspace is a subset of the metaverse. The metaverse is hierarchical and every model in the metaverse has a unique path. Models are visible to the authorised user.

With these design, we developed the kernel of AToMPM, called Ark. This tool is a new version for ArkM3. It has basic modelling functionality such as model creation, searching, and conformance checking. Ark also has an interpreter to execute action models. At last, we presented a case study of Readers/Writers system Petri Net model.

6.1 Attemps to Improve the Performance

We run some performance tests on the Ark. Unfortunately, the speed was not satisfactory. When ArkM3 was load into memory as Himesis graph , it took the kernel 1 second to create a model of ten ArkM3 class instances. When ArkM3 was the compiled Python module, the speed was 100 time faster.

We used Python *cProfile* package to profile the program performance. After the profiling creating process, we found that the speed was compromised for the following reasons.

1. Loading ArkM3 into memory is very time-consuming.
2. To create instances of one definition, Ark repeatedly traverses the same metamodel element.
3. Himesis has many string splits and joins operation when dealing with the paths.
4. An instance contains too many graph node because we represent everything, from a class to a data as graph.
5. Processing graph node is slower than compiled modules because there are a lot checking and comparisons in Himesis.

6. The scripting language Python is not as efficient as that of C++ or C.

We have made some efforts to accelerate Ark.

Lazy instantiation

Lazy instantiation will not generate everything at the first place. Instead, an object is instantiated only when it is being used. Creating an instance of the class `A` is to create an empty object `a:A`. Only when the attributes `name` are assigned a value, should the system look up the metamodel and create the attribute `name` and add it to the object `a:A`. This significant improved the speed of creation. However, it achieved this at the cost of more traversing during model manipulation. For this reason, it is not a good solution to our problem.

XML loader

To load a ArkM3 graph model in memory, we need to first using Python modules to create an ArkM3 in memory and then use this ArkM3 to bootstrap an ArkM3 graph model. This is very time-consuming. We tried to use Himesis XML loader hoping it will make loading ArkM3 faster. XML loader did bring us certain portability. However, Himesis XML serialization tool was not sufficient for our problem. The depth of Himesis graph in a class model is deep and Himesis loader involved a large number of file I/O. The loading process was even 100 times slower than the old approach.

Less Python string operation

After profiling the creation process, we found that five Himesis methods (`_resolvePath` which analyses the string name and gets a sub node in a node using relative path, `isDescendantOf` which examines if a node is the descendant of another, `hasElement` which examines if a node has a sub node, `getParent` which gets the immediate parent of a node, and `getElement`) are heavily used. The time spent on these methods is outstanding that they took almost half the execution time. The five are all related to `_resolvePath`, that splits the global id string into segments so as to traverse the model hierarchy.

In Python, string split is time-consuming. We have tried several ways to avoid it. Finally, we adopted the hash table.

1. Using hash table: This method has improved the performance by $2\tilde{3}$ times.
2. Using list: Using list requires less splits but more concatenation and more memory is required for lists.
3. Using pre compilation: This method resolve all the references before the model is executed. However it is more useful for action interpretation, but not very helpful in creation.

Abandon the deep Himesis graphs

The profile shows that creating a Graph node is also time-consuming. Everything in AToMPM is implemented as graph node from a package to a primitive type and value. Consequently, the model is very deep and the number of nodes are large. Nevertheless, this is tolerable because this is the way how AToMPM clearly and completely represents a model.

Himesis is well-defined and tested. To make sure the model is safe, it has many conditions statement in the source code. To make the code more modular, it includes many function calls. This could be another reasons why processing Himesis graph models is slow.

6.2 Future improvement on graph structure

Eventually there are no single method that can solve all the problems. Improving the performance by hundred times mainly depends on upgrading underlying structure of Himesis.

At the same time, the action models are completely defined in Himesis graph. It also cause troubles in implementing the action interpreter. The interpreter have to use compiler technique to work on the very basic functionality such as maintaining symbol table of an action and managing the stack for function calls. Consequently, AToMPM did not benefit from the up-to-date compiler technology. Therefore, we need to find a better graph metamodel.

6.3 Some Other Future Work

Further refining primitives would be useful. To do this, we will need to distribute the beta version of AToMPM and receive feedbacks from the users. In addition, building complex operators using the simple primitives will be very useful for a user-welcome interface.

AToMPM also allows library, as supported by its metaverse, so that developing bigger and more useful functions would be important for modelling and transformation. Model importation and merge are important for implemention of such a library system.

Serialization is an important function to make the models portable. Currently, we are using Himesis XML generator and loader. However, as mentioned earlier, the loader is not suitable for large scale task. We tend to find a widely used schema for serialization, so that we can ease distribution of AToMPM.

Besides improving the kernel performance, we have been discussing some further development of AToMPM. Adding graphical concrete syntax could provide better API for the modellers. Running AToMPM on a server and providing web modelling application is an interesting and welcoming development in the future.

Bibliography

[Á01] José Álvarez. MML and the metamodel architecture. In *WTUML: Workshop on Transformation in UML 2001*, 2001.

[AGK09] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, 2009.

[AK01] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, number 2185 in LNCS, pages 19–33, 2001.

[AK02] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transaction on Modeling and Computer Science*, 12(4):290–321, October 2002.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7:345–359, 2008. 10.1007/s10270-007-0061-0.

[AKN+06] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi, and Attila Vizhanyo. The design of a language for model transformations. *Software and Systems Modeling*, 5:261–288, 2006. 10.1007/s10270-006-0027-7.

[BSME03] Frank Budinsky, Dave Steinberg, Ed Merks, and Ray Ellersick. *Eclipse Modeling Framework: A Developer's Guide*, chapter Introducing EMF, pages 11–40. The Eclipse Series. 2003.

[CEKI00] Tony Clark, Andy Evans, Stuart Kent, and IBM. A feasibility study in rearchitecting UML as a family of languages using a Precise OO meta-modeling approach. Technical report, Precise UML Group, 2000. MML.

[CL08] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*, chapter Petri Nets, pages 223–268. SpringerLink Engineering. Springer Science+Business Media, 2008.

[Coi87] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *In Object Oriented Programming Systems Languages and Applications*, pages 156–162, 1987.

[CSW08] Tony Clark, Paul Sammut, and James Willans. *Applied Metamodelling: A Foundation For Language Driven Develeopment*. Ceteva, 2nd edition edition, 2008.

[DFV+09] Z. Drey, F. Fleurey, D. Vojtisek, C. Faucher, and Vincent Mahé. *Kermeta Language, Reference Manual*, 2009.

[DG06] Stephane Ducasse and Tudor Girba. Using Smalltalk as a reflective executable metalanguage. In *MODELS2006*, volume 4199 of *LNCS*, pages 604–618. Springer-Verlag, 2006.

[dLG10] Juan de Lara and Esther Guerra. Deep meta-modelling with MetaDepth. *TOOLS*, (48):1–20, 2010.

[dLJVM03] Juan de Lara Jaramillo, Hans Vangheluwe, and Manuel Alfonseca Moreno. Using metamodelling and graph grammars to create modelling environments. *Electronic Notes in Theoretical Computer Science*, 72(3):36 – 50, 2003. GT-VMT'2002, Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation).

[dLV02] Juan de Lara and Hans Vangheluwe. AToM3: A Tool for Multi-formalism Meta-Modelling. In *Fundamental Approaches to Software Engineering*, pages 174–188. Springer Berlin / Heidelberg, 2002.

[EGdL+05] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. 2005.

[emf] Eclipse Modeling Framework Project (EMF) at http://www.eclipse.org/modeling/emf/.

[ERW08] Jürgen Ebert, Volker Riediger, and Andreas Winter. Graph technology in reverse engineering: The TGraph approach. In *Workshop Software Reengineering*, pages 67–81, 2008.

[Fla02] Rony G. Flatscher. Metamodeling in EIA/CDIF—meta-metamodel and metamodels. In *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, volume 12, pages 322 – 342, 2002. CDIF, 4 layered architecture is from IRDS, CDIF and MOF.

[FUM08] Semantics of a foundational subset for executable UML models (fUML) Version 1.0 - Beta 1, November 2008.

[GHJV95] Erich Gamma, Rechard Helm, Ralph Johnson, and John Vilssides. *Design Patterns: Elements of Reusable Object Oriented Software*. Pearson Education, 1995.

[GKD01] A. Goh, Y. K. Koh, and D. S. Domazet. ECA rule-based support for workflows. *Artificial Intelligence in Engineering*, 15(1):37 – 46, 2001.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc, 1983.

[GSZ04] Holger Giese, Andy Schürr, and Albert Zündorf, editors. *Fujiba Days*, TU Darmstadt, Germany, 15th -17th September 2004.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987.

[Hec06] Reiko Heckel. Graph transformation in a nutshell. *Electron. Notes Theor. Comput. Sci.*, 148:187–198, February 2006.

[HHMM00] I. Herman, I. Herman, M. S. Marshall, and M. S. Marshall. GraphXML - an XML based graph interchange format. Technical report, 2000.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26:287–313, June 1996.

[Him96] Michael Himsolt. GML: A portable graph file format. Technical report, Universitt Passau, 94030 Passau, Germany, 1996.

[Hof99] Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 20 years edition edition, 2 1999.

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. *Technical report: MCS00-16 , Jerusalem, Israel*, 2000.

[HWS00] R. C. Holt, A. Winter, and Andy Schurr. GXL: Toward a standard exchange format. In *Dagstuhl Seminar Interoperability of Reengineering Tools*, pages 162–171, 2000.

[Ins89] American National Standard Institute. Information Resource Dictionary System (IRDS), 1989.

[JB06] Frederic Jouault and Jean Bezivin. KM3: A DSL for metamodel specification. In R. Gorrieri and H. Wehrheim, editors, *FMOODS 2006*, volume 4037 of *LNCS*, pages 171–185, 2006.

[JK06] Frédéric Jouault and Ivan Kurtev. On the architectural alignment of ATL and QVT. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1188–1195. ACM, New York, NY, USA, 2006.

[KÖ5] Alexander Königs. Model transformation with triple graph grammars. In *In Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego*, 2005.

[KÖ6] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5(4):369–385, December 2006.

[Ker09] Kermeta. Kermeta - breathe life into your metamodels at http://www.kermeta.org/, 2009.

[KPG07] Pierre Kelsen, Elke Pulvermueller, and Christian Glodt. A declarative executable language based on OCL for specifying the behavior of platform-independent models. In *Ocl4All 2007 Workshop, Nashville*, 2007.

[KPG08] Pierre Kelsen, Elke Pulvermueller, and Christian Glodt. Specifying executable platform-independent models using ocl. *ECEASST*, 9, 2008.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *In: Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006*, pages 128–142. Springer, 2006.

[Kuh89] D.L Kuhn. Selecting and effectively using a computer aided software engineering tool. In *Annual Westinghouse computer symposium*, pages 6–7, 1989.

[LLMC05] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science*, 127(1):65 – 75, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[LV07] Holger Gieseand Tihamér Levendovszky and Hans Vangheluwe. Summary of the workshop on multi-paradigm modeling: Concepts and tools. In Thomas Kühne, editor, *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 252–262. Springer Berlin / Heidelberg, 2007.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*, chapter The Visitor Family of Design Patterns, pages 525–558. Prentice Hall, 2002.

[MBC+94] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling With Generalised Stochastic Petri Nets*, chapter Petri Net and Their Properties, pages 31–62. Wiley Series in Parallel Computing. John Wiley and Sons, 1994.

[mda03] MDA Guide Version 1.0.1, 6 2003.

[MFJ05] P. Muller, F. Fleurey, and J. Jezequel. Weaving executability into Object-Oriented meta-languages. In L. Briand, editor, *Proceedings of MODELS/UML'2005*, pages 264–278, Montego Bay, Jamaica, 2005. S. Kent. Springer.

[MG05] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. 2005.

[MOF02] Meta Object Facility (MOF), Version1.4, 4 2002.

[MOF06] Meta Object Facility (MOF) core specification Version 2.0, January 2006.

[MSD09] MSDL. AToM3 at http://atom3.cs.mcgill.ca/, 2009.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frederic Fondement, and Jean Bezivin. Platform independent web application modeling and development with Netsilon. *Software and Systems Modeling*, 4:424–442, 2005. Xion.

[MV02] Pieter J. Mosterman and Hans Vangheluwe. Guest editorial: Special issue on computer automated multi-paradigm modeling. In *ACM Transactions on Modeling and Computer Simulation*, volume 12, pages 249–255, October 2002.

[OCL97] Object Constraint Language specification Version 1.1, September 1997.

[OCL06] Object Constraint Language specification Version 2.0, May 2006.

[Pro05] Marc Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis, School of Computer Science McGill University, Montreal, Canada, 2005.

[QVT08] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0, 4 2008.

[qvt09] Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Version 1.1 - Beta 2, December 2009.

[RK09] Arend Rensink and Jan-Hendrik Kuperus. Repotting the geraniums: On nested graph transformation rules. In Tiziana Margaria, Julia Padberg, and Gabriele Taentzer, editors, *Electronic Communications of the EASST Pre-proceedings*, 2009.

[sC76] Peter Pin shan Chen. The Entity-Relationship Model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[Sch97] A. Schfürr. *Programmed Graph Replacement Systems*, pages 479–546. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[SG08] Markus Stumptner and Steve Georg Grossmann. Introduction to domain modelling environment at http://www.cis.unisa.edu.au/ cisgg/wiki/dome/index.html, 2008. last updated 2008-07-08.

[SGJ02] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Using UML Action Semantics for model execution and transformation. *Information Systems*, 27(6):445 – 457, 2002.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. MetaEdit+ a flexible graphical environment for methodology modelling. pages 168–193, 1991.

[Syr09] Eugene Syriani. MoTif: The modular timed graph transformation language. Invited Talk at Universidad Autonoma de Madrid. Madrid, 2009.

[Tae00] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance*, AGTIVE '99, pages 481–488, London, UK, 2000. Springer-Verlag.

[Tha00] Bernhard Thalheim. *Entity-relationship Modeling: Foundations of Database Technology*, chapter Extending the Entity-Relationship Model, pages 55–104. Springer, 2000.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg, 2009.

[TYF86] Toby J. Teorey, Dongqing Yang, and James P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv.*, 18:197–222, June 1986.

[UML03] OMG Unified Modeling Language specification Version 1.5, 2003.

[Van08] Hans Vangheluwe. Proceedings of the seventh international workshop on graph transformation and visual modeling techniques: Foundations of modelling and simulation of complex systems. *Electronic Communications of the EASST*, 10, 2008.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68:187–207, October 2007.

[vmt] VMTS at http://www.aut.bme.hu/Portal/Vmts.aspx?lang=en.

[VSB07] Hans Vangheluwe, Ximeng Sun, and Eric Bodden. Domain-specific modelling with AToM3. *ICSOFT (PL/DPS/KE/MUSE)*, pages 298–304, 2007.

[WKR01] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In S. Diehl, editor, *Software Visualization: International Seminar Dagstuhl Castle*. Springer Verlag, 2001.

[YBJ01] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *SIGPLAN Not.*, 36:50–60, December 2001.

[ZKP00] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.

A

The Mapping Between ArkM3 and Himesis

We will show how ArkM3 instances are mapped to Himesis models.

DataType and DataValue to Himesis Node

DataType and PrimitiveValue are special classes in Himesis. They are the basic type of Node. They can not contain any other nodes. The following graphs introduce the mapping of DataType and DataValue.

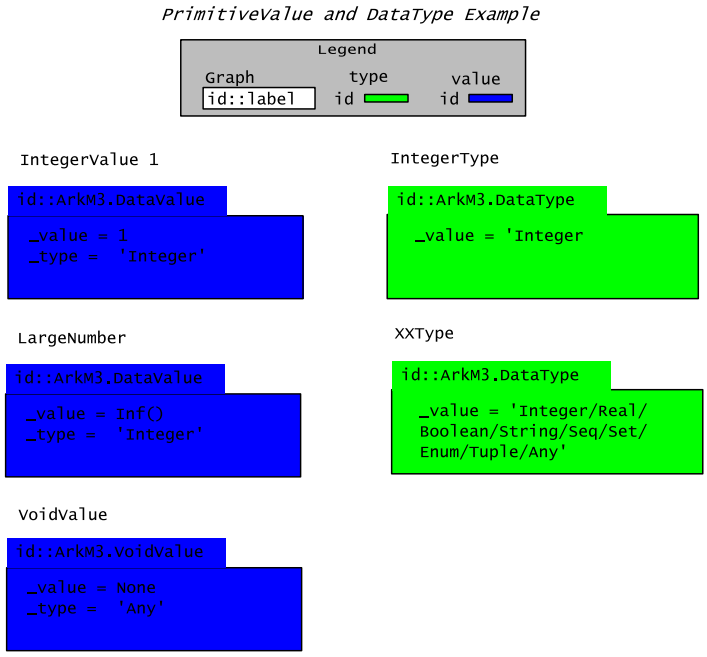


Figure A.1: Map ArkM3 Primitive Data Values to Himesis

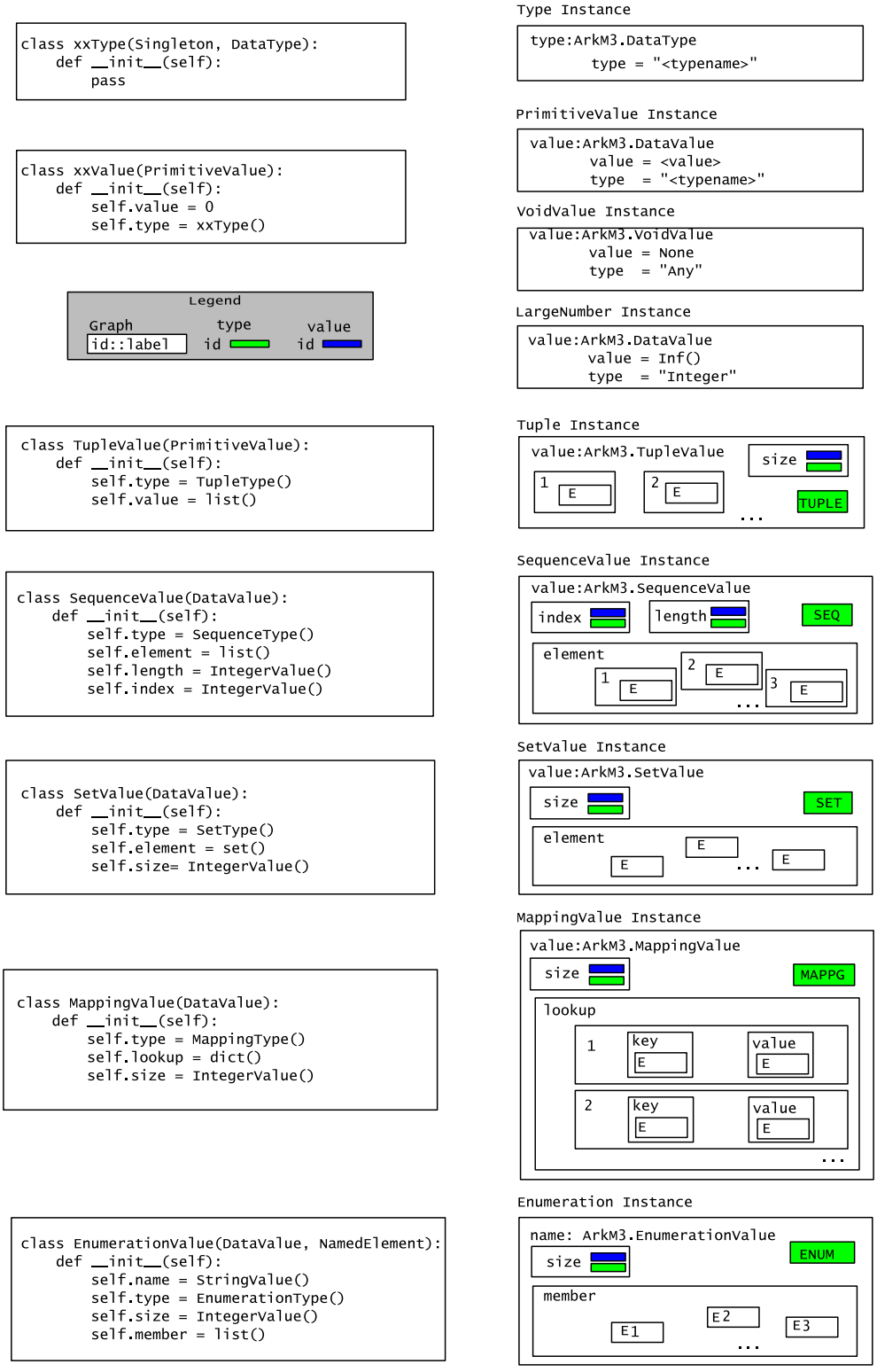


Figure A.2: Map ArkM3 Complex Data Values to Himesis

Map Action Language Elements to Himesis Graph

This section shows how the action language models are mapped to Himesis.

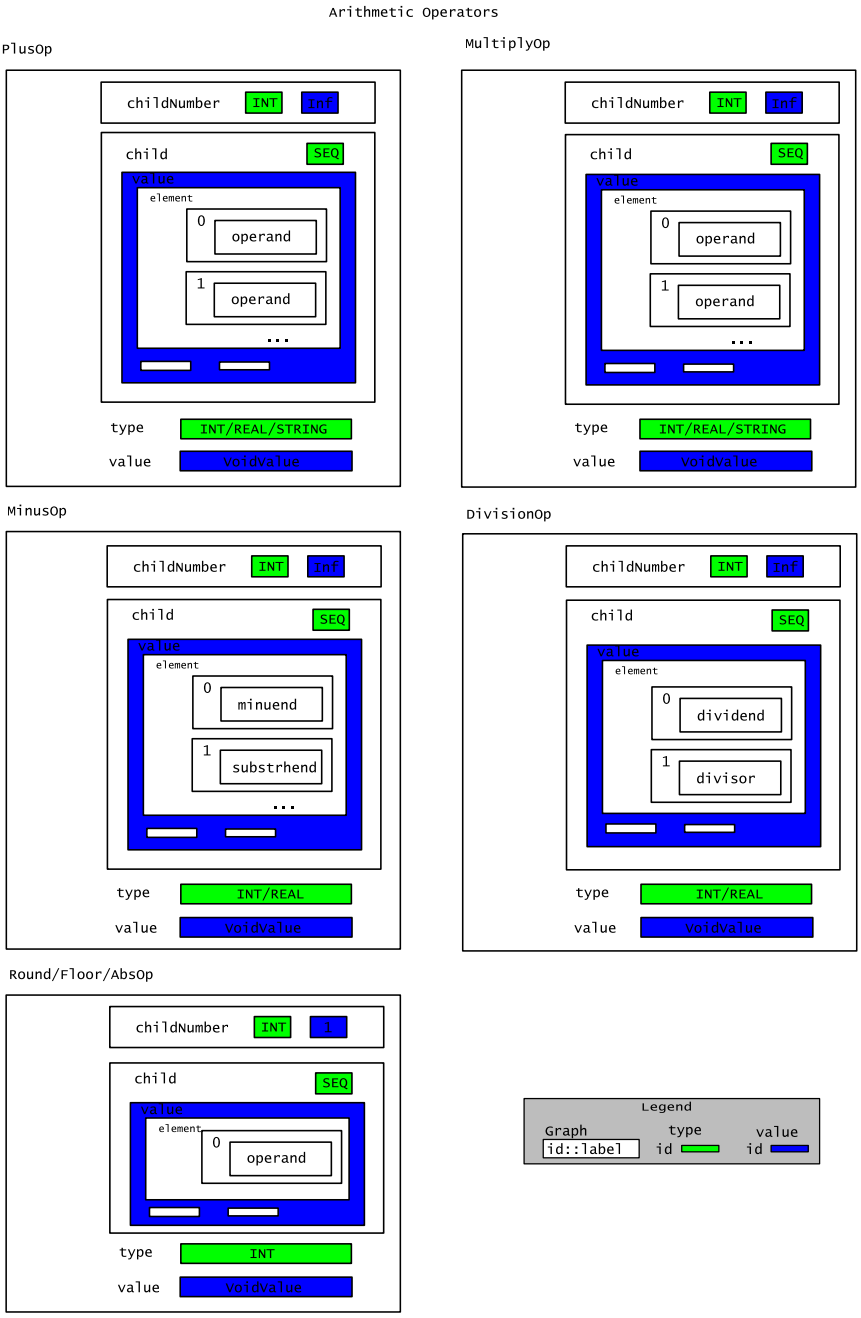


Figure A.3: Map ArkM3 Operators to Himesis. 1

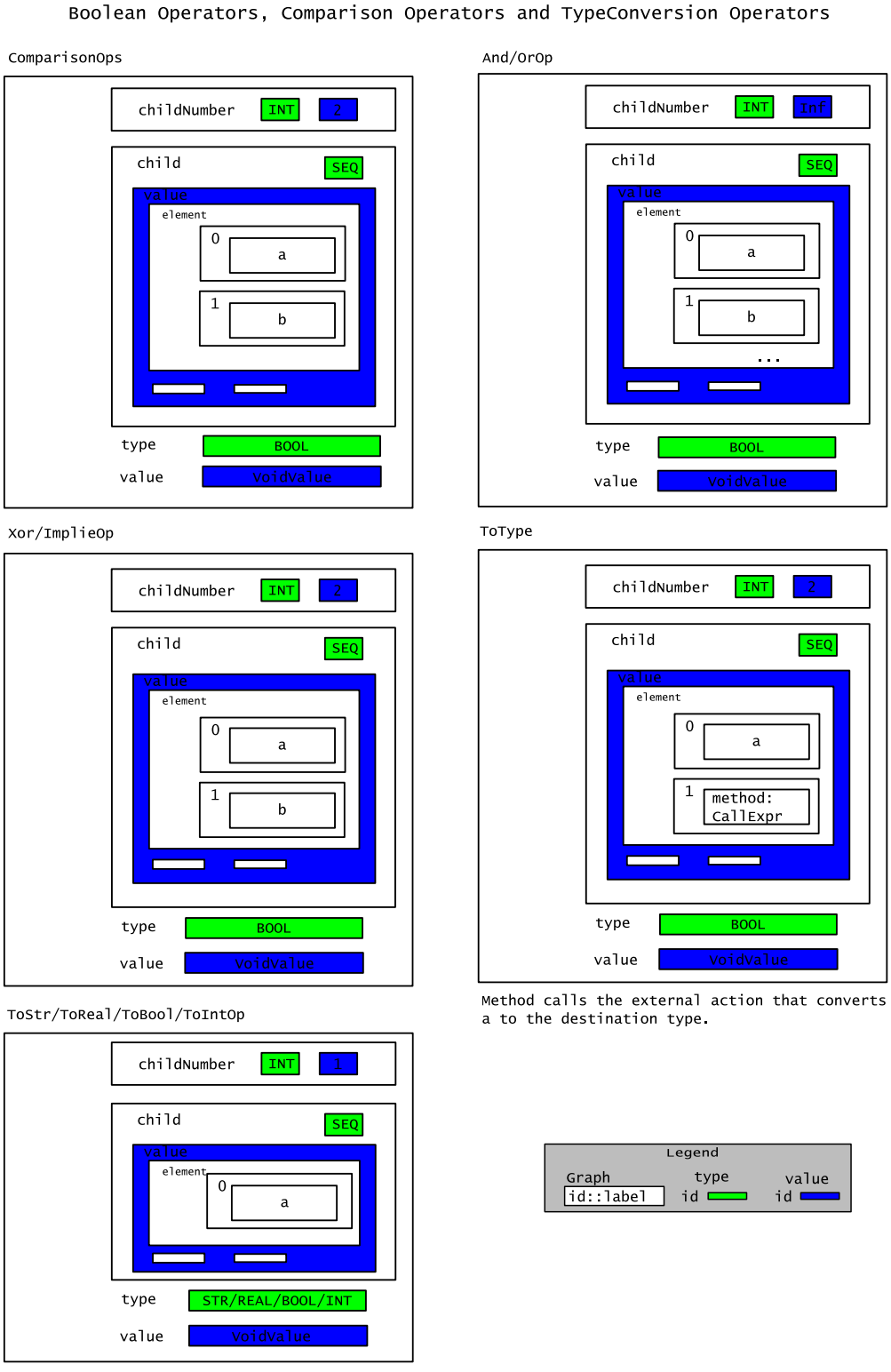


Figure A.4: Map ArkM3 Operators to Himesis. 2

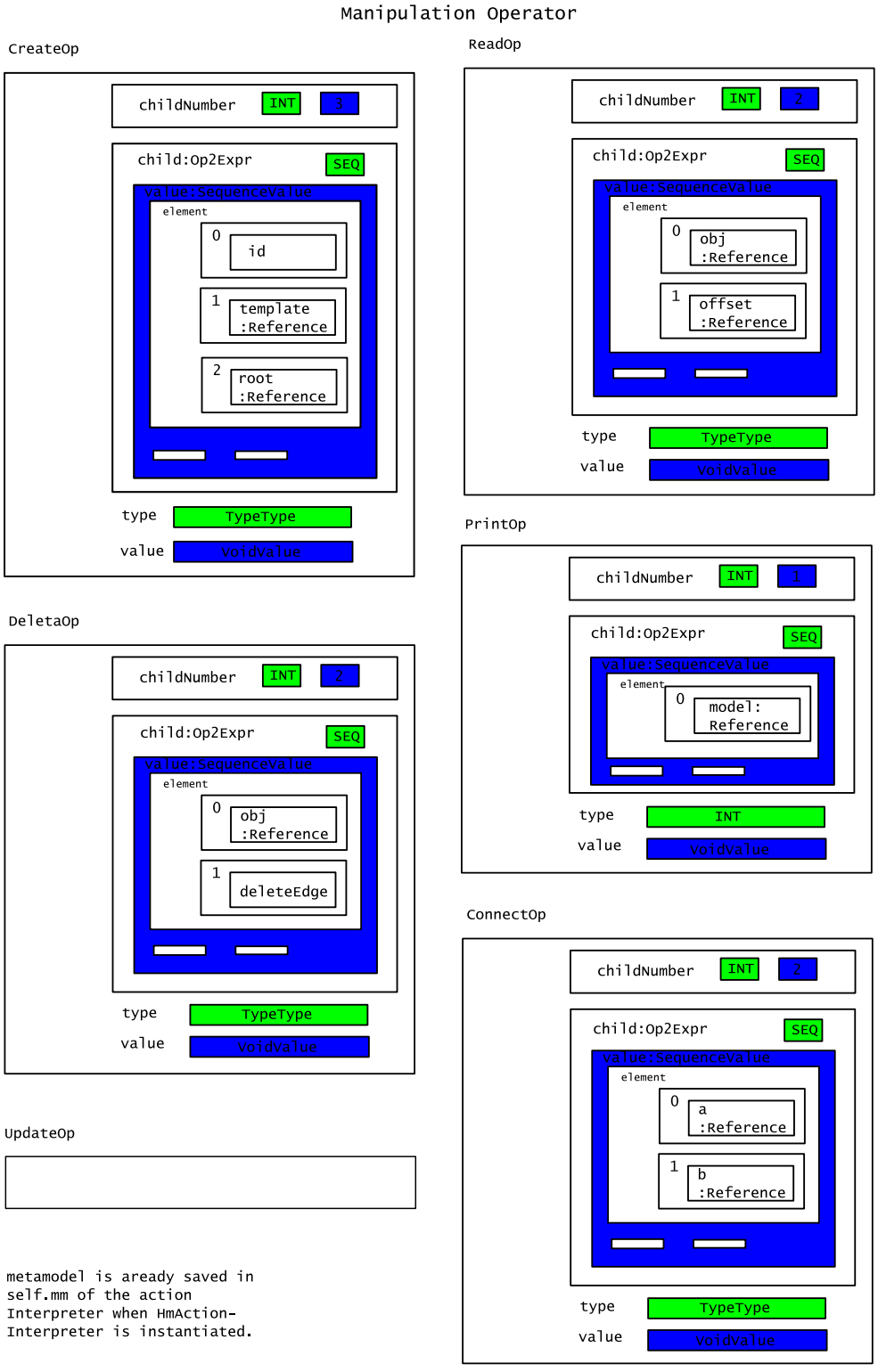


Figure A.5: Map ArkM3 Operators to Himesis. 3

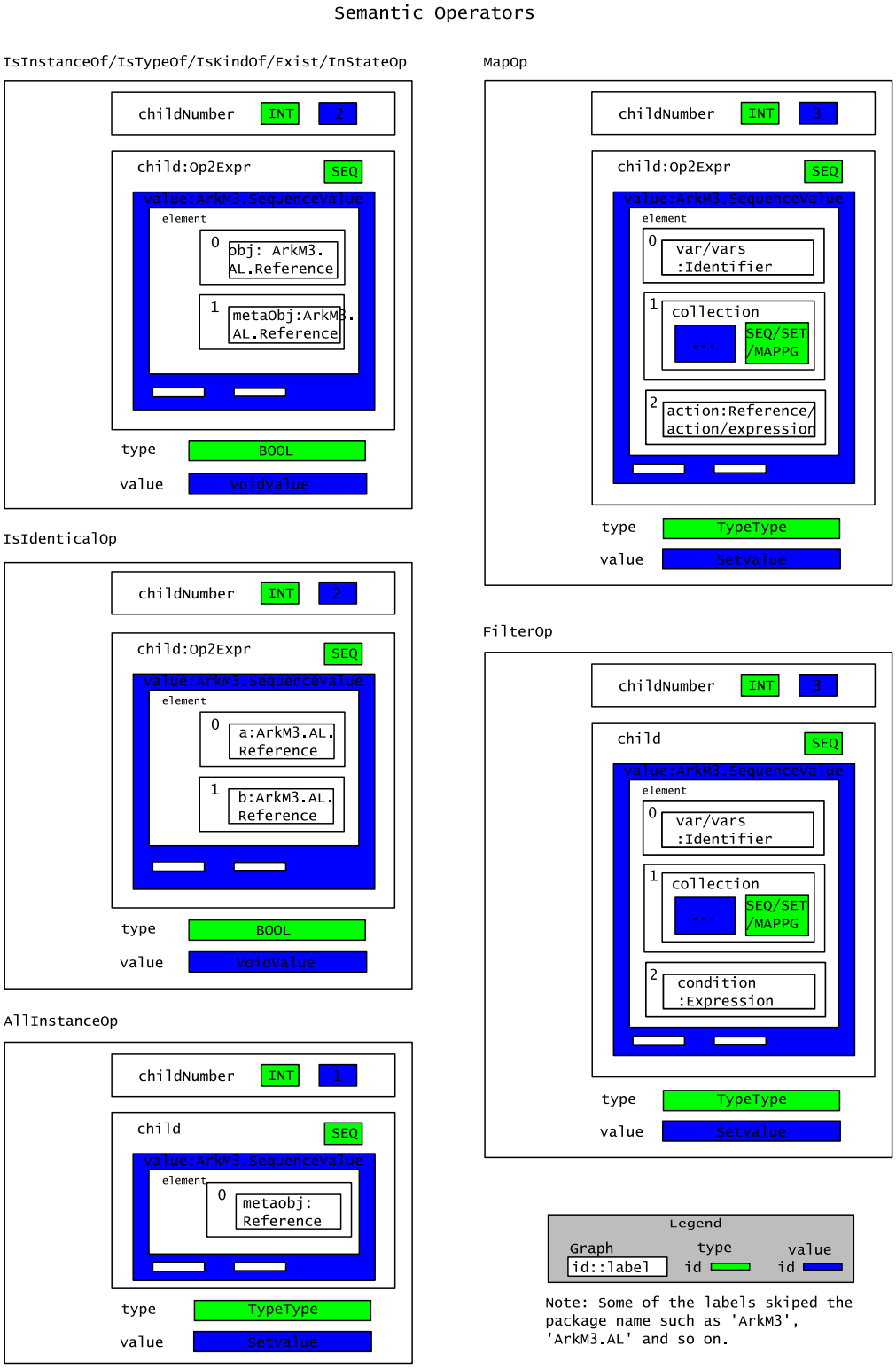


Figure A.6: Map ArkM3 Operators to Himesis. 4

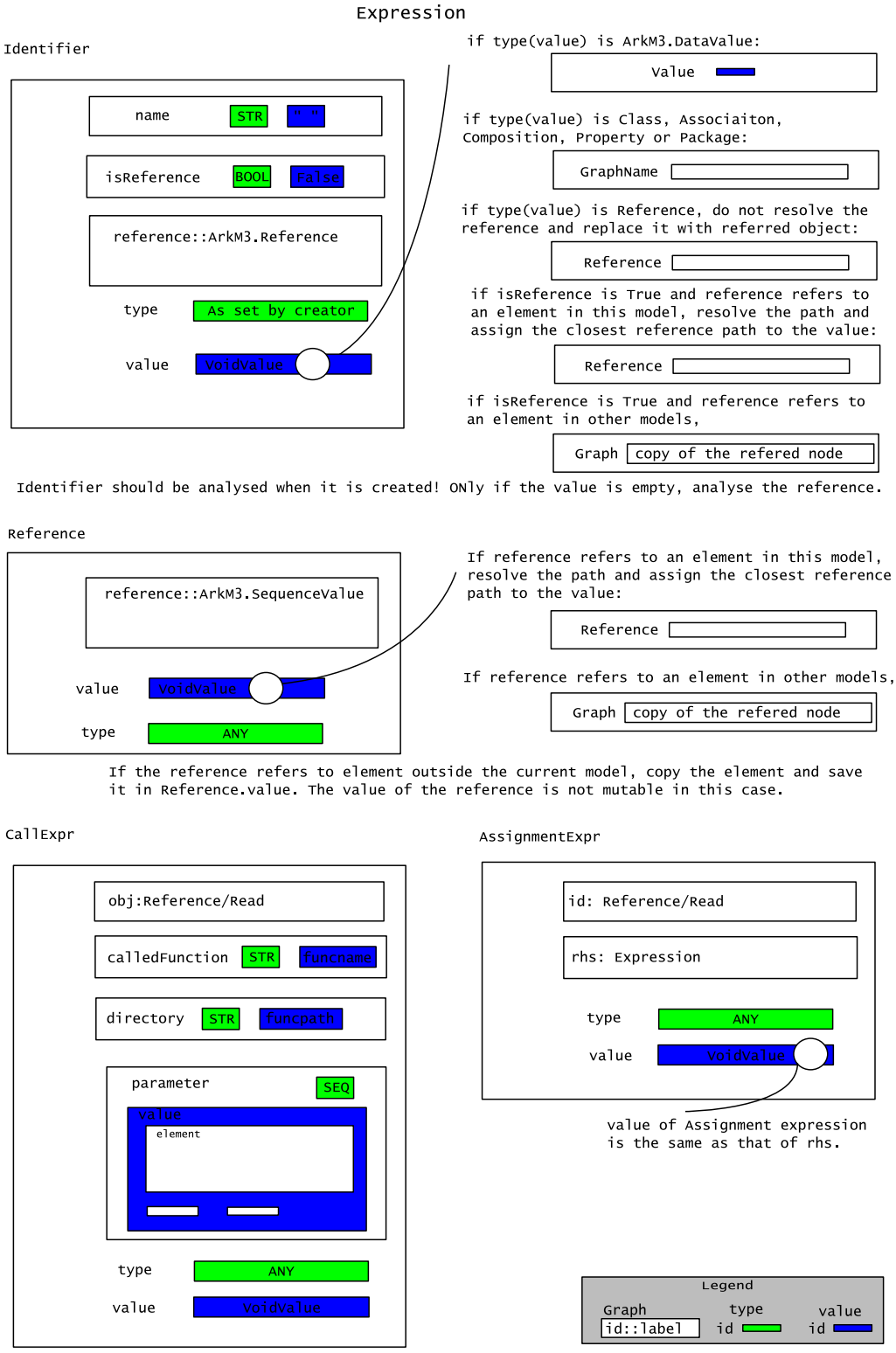


Figure A.7: Map ArkM3 Expressions to Himesis

Map Class Diagram instances to Himesis

Package Definition (Psuedo Code)

```
P1 = factory.createPackage("Package A")
P1.add(createClass("Class_A", isAbstract=False))
P1.add(createAssociation("Association_A", isAbstract=False))
P1.add(createAction("Action_A", isimplemented=False, parameter=[], returnType=Int))
```

Package instance represented by Himesis

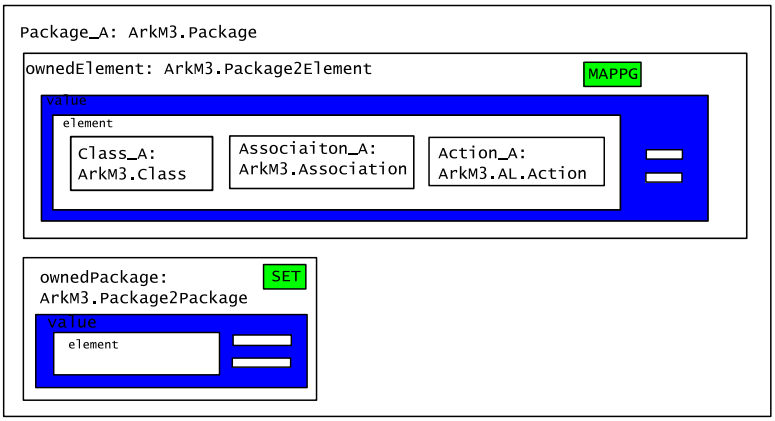


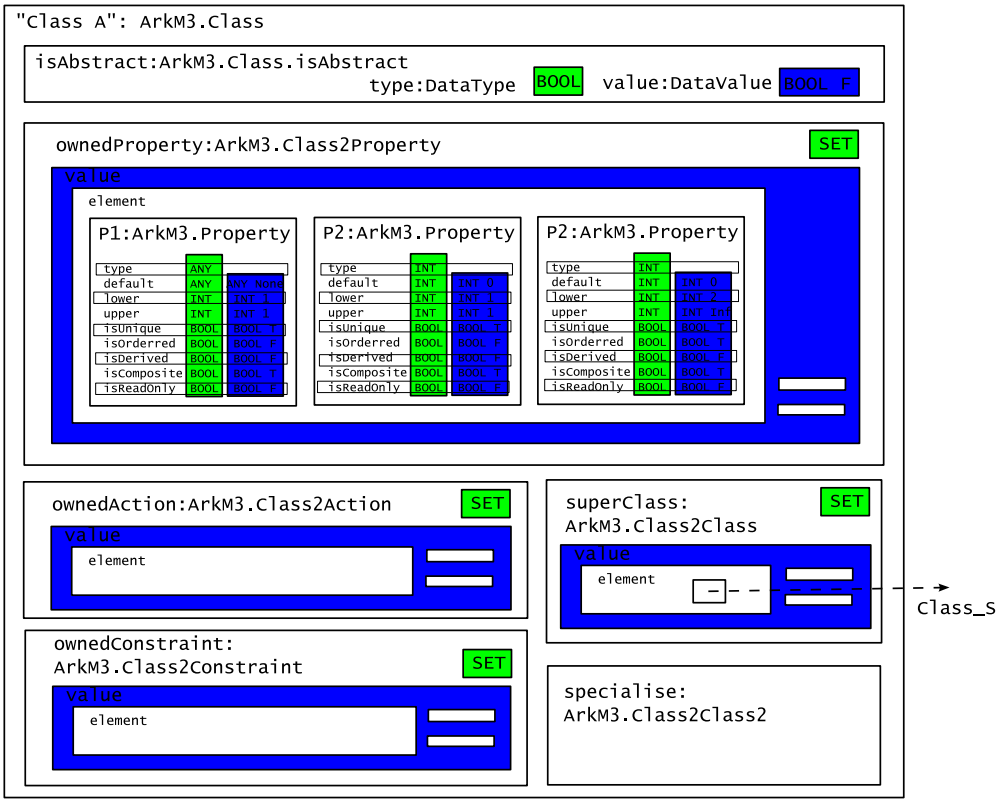
Figure A.8: Map ArkM3 Package to Himesis

Class Definition

```
C1 = factory.createClass("Class A", isAbstract = False)
C1.addProperty(createProperty("P!", lower=1, upper=1))
C1.addProperty(createProperty("P2", lower=1, upper=1, type=Int, default=0))
C1.addProperty(createProperty("P3", type=Int, isOrdered=True, lower=2, upper=*))

C1.addSuperClass(Class_S) !Class_S does not have one attribute, pS.
```

The model equals this model in Himesis Graph Node



Instance that is represented by Himesis Graph Node

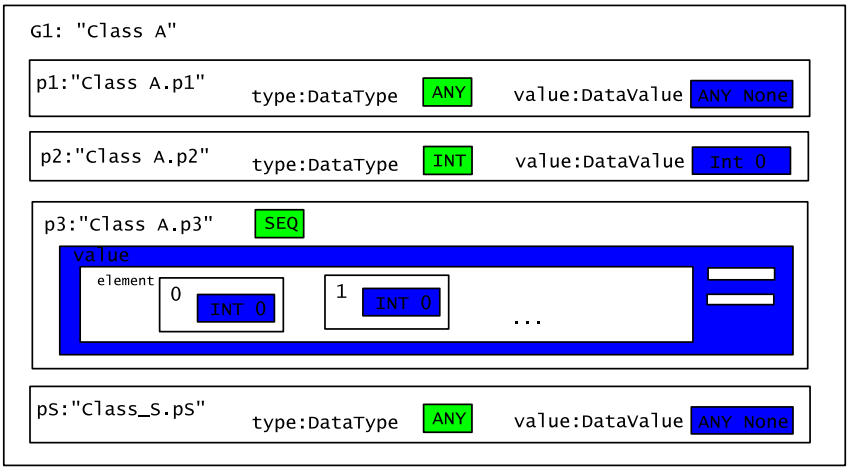
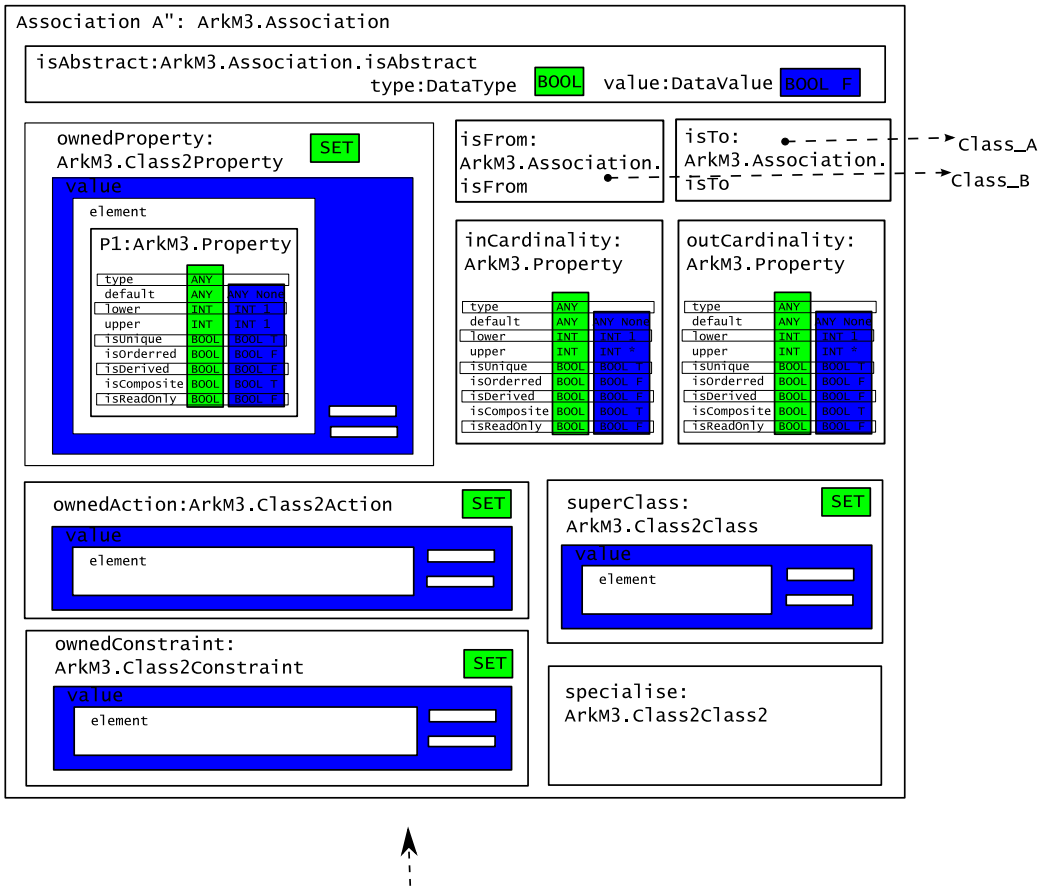


Figure A.9: Map ArkM3 Class to Himesis

Class Definition (Psuedo Code)

```
A1 = factory.createAssociation("Association A", isAbstract = False)
A1.isFrom = Class_A
A1.isTo = Class_B
A1.addProperty(createProperty("inCardinality",isComposite=True, isUnique=True,\
isOrdered=False,lower=1, upper=*))
A1.addProperty(createProperty("outCardinality", isComposite=True, isUnique=True,\
isOrdered=False, lower=1, upper=*))
A1.addProperty(createProperty("p1", lower=1, upper=1, type=Int, default=0))
```

The model equals this model in Himesis Graph Node



Instance that is represented by Himesis Graph Node

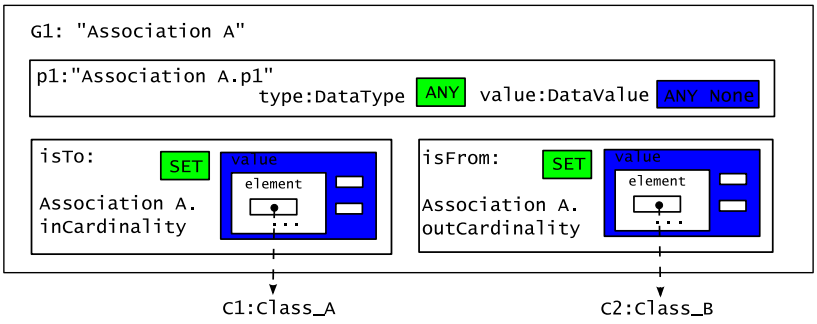


Figure A.10: Map ArkM3 Association to Himesis

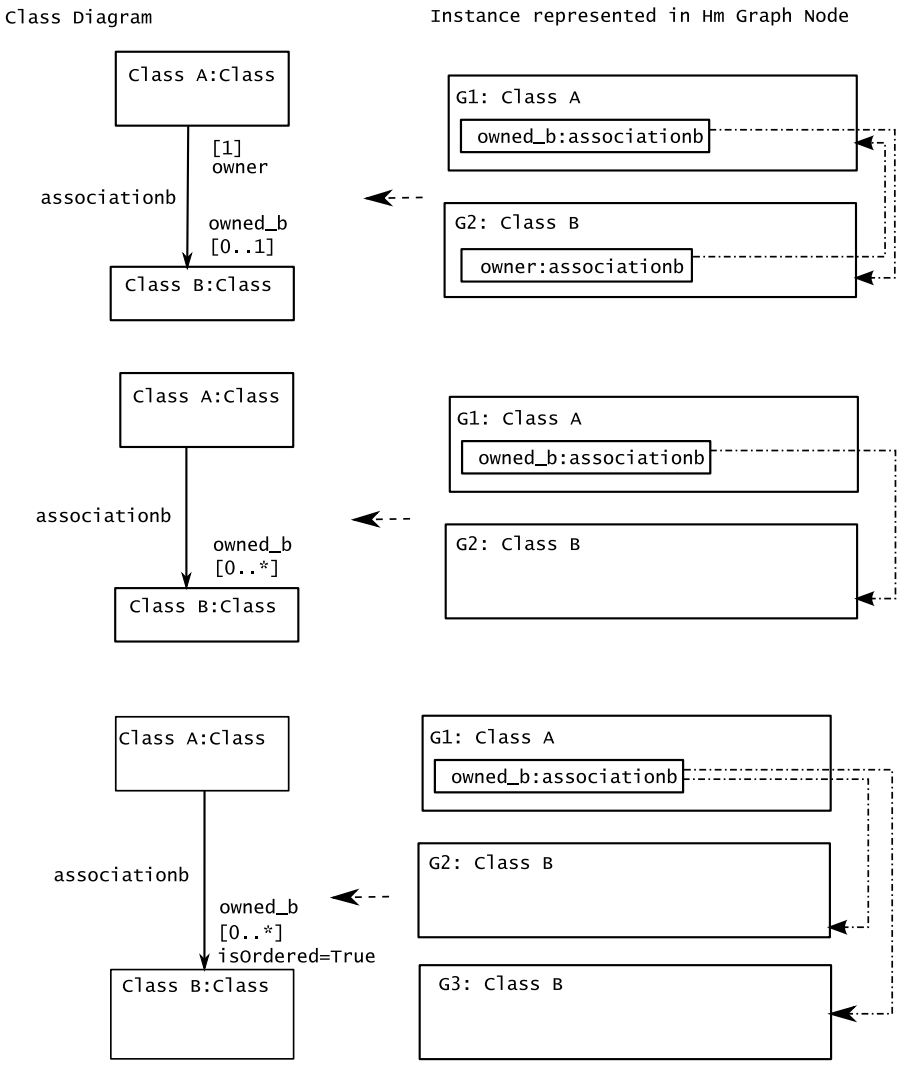


Figure A.11: Map ArkM3 Class Instances and their Associations to Himesis

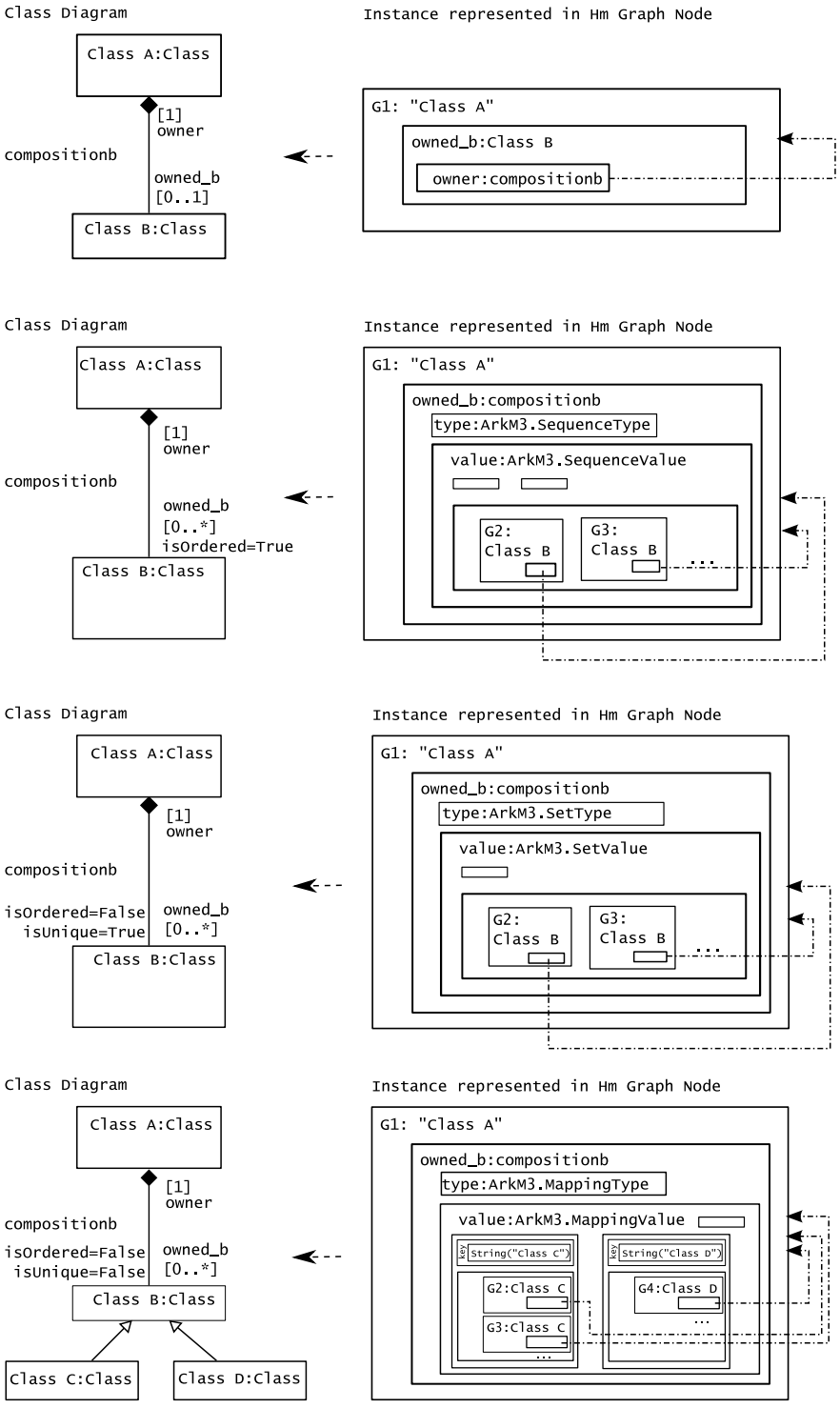


Figure A.12: Map ArkM3 Class Instances and their Compositions to Himesis

B

Define CCPN Metamodel, Python Code

```
from Himesis.Value import Value
from Himesis.VoidValue import VoidValue
from Himesis.PrimitiveValue import PrimitiveValue
from Himesis.Type import Type
from Himesis.Graph import Graph
from Himesis.infinity import Inf

from Metaverse import METAVERSE

from function.HmHmFactory import HmHmFactory
from function.ModelCreator import ModelCreator

#private functions to create Himesis nodes
def _node(id,label):
    return Graph(id, label)
def _value(id, label, value, type):
    return Value(id, label, value, type)
def _voidvalue(id, label):
    return VoidValue(id, label, None, "ANY")
def _type(id, label, value):
    return Type(id, label, value)


# metamodel of CCPN
def createCCPN_META():
    metamodel = ModelCreator().createPackage("CCPetriNet_META_Hm", None)
    METAVERSE.add(metamodel)
    factory = ModelCreator(root = metamodel)

    #copy ArkM3.Element, abstract class
    superClass = factory.createClass("Element", root = metamodel["ownedElement.value.element"], isAbstract=True, package=metamodel)
    #the properties of the class instance are saved under the offset "ownedProperty.value.element"
    factory.createProperty("ownedComment", root = superClass["ownedProperty.value.element"], lower=0, upper=Inf(), hostClass = superClass)
    factory.createProperty("isAccessable", root = superClass["ownedProperty.value.element"], hostClass = superClass)
    factory.createProperty("modelVersion", root = superClass["ownedProperty.value.element"], hostClass = superClass)
    #increase the object counter in the package, the number is saved in graph node with the offset "ownedElement.value.size.value"
    metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)


    #copy ArkM3.Package
    pack = factory.createClass("Package", root = metamodel["ownedElement.value.element"], package=metamodel)
    #the next two lines indicate Package inherits from Element
    factory.add_to_set(pack["super.value"], _node("1", "Himesis.Graph"))
    metamodel.connect(pack["super.value.element.1"], superClass)
    factory.createProperty("metaInfo", root = pack["ownedProperty.value.element"], typ="STRING", default = "", hostClass = pack)
    metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)

    #Package connects to Elements by a Composition instance, meaning Package can contain anything.
    compo = factory.createComposition("pack2elem", root = metamodel["ownedElement.value.element"], isComposite=True, \
        isUnique=True, isOrdered=False, package=metamodel, isFrom=pack, isTo=superClass, end1="package", end2="ownedElement")
    factory.add_to_set(compo["super.value"], _node("1", "Himesis.Graph"))
    metamodel.connect(compo["super.value.element.1"], superClass)
    factory.createProperty("inCardinality", root = compo, isComposite=True, isUnique=True, isOrdered=False, lower=1, upper=1)
    factory.createProperty("outCardinality", root = compo, isComposite=True, isUnique=True, isOrdered=False, lower=1, upper=Inf())
    metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)

    #create CCPetriNet_META_Hm.Place Class
    place = factory.createClass("Place", root = metamodel["ownedElement.value.element"], package=metamodel)
    factory.add_to_set(place["super.value"], _node("1", "Himesis.Graph"))
    metamodel.connect(place["super.value.element.1"], superClass)
    factory.createProperty("token", root = place["ownedProperty.value.element"], typ="INTEGER", isComposite=True,lower=1, \
        upper=1, default = 0, hostClass = place)
```

```
factory.createProperty("maximumToken", root = place["ownedProperty.value.element"], typ="INTEGER", isComposite=True, \
    lower=1, upper=1, default = Inf(), hostClass = place)

#
# Add constraint to the CCPetriNet.Place
const1 = factory.createConstraint("maximumTokenConstraint", root=place["ownedConstraint.value.element"], \
    condition="CONDITION", isImplemented = True, hostElement = [place])
lhs = factory.createIdentifier("t", isRef = True, reference = factory.createReference(id="ref", ref=["SELF","token"], \
    meta= "ArkM3.AL.IdentifierReference"))
rhs = factory.createIdentifier("mt", isRef = True, reference = factory.createReference(id="ref", ref=["SELF","maximumToken"],\
    meta="ArkM3.AL.IdentifierReference"))
#print const1["expression"]
factory.createNotGreaterThan(root = const1["expression"], child = [lhs,rhs])
# Add constraint to the CCPetriNet_META_Hm.Place
# The process is in the other method called _create_Constraint1()
metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)


#create CCPetriNet_META_Hm.Transition
trans = factory.createClass("Transition", root=metamodel["ownedElement.value.element"], package=metamodel)
factory.add_to_set(trans["super.value"], _node("1", "Himesis.Graph"))
metamodel.connect(trans["super.value.element.1"], superClass)
metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)

#create association  CCPetriNet_META_Hm.Place2Transition which connect from p1 to tr
pl2tr = factory.createAssociation("Place2Transition", root=metamodel["ownedElement.value.element"],  isUnique=True, \
    isOrdered=False, package=metamodel, isFrom = place, isTo = trans, end1="last", end2="next")
factory.add_to_set(pl2tr["super.value"], _node("1", "Himesis.Graph"))
metamodel.connect(pl2tr["super.value.element.1"], superClass)
factory.createProperty("inCardinality", root = pl2tr, isComposite=True, isUnique=True, isOrdered=False, lower=1, upper=Inf(), hostClass=pl2tr)
factory.createProperty("outCardinality", root = pl2tr, isComposite=True, isUnique=True, isOrdered=False, lower=1, upper=Inf(), hostClass=pl2tr)
factory.createProperty("weight", root = pl2tr["ownedProperty.value.element"], isComposite=True, lower=1, upper=1, typ="INTEGER",\
    default=0, hostClass=pl2tr)
metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)

#create association from tr to pl
tr2pl = factory.createAssociation("Transition2Place", root=metamodel["ownedElement.value.element"], isUnique=True,\
    isOrdered=False, package=metamodel, isFrom =trans, isTo =place, end1="last", end2="next")
factory.add_to_set(tr2pl["super.value"], _node("1", "Himesis.Graph"))
metamodel.connect(tr2pl["super.value.element.1"], superClass)
factory.createProperty("inCardinality", root=tr2pl, isComposite=True, isUnique=True, isOrdered=False,lower=0, upper=Inf(), hostClass=tr2pl)
factory.createProperty("outCardinality", root=tr2pl, isComposite=True, isUnique=True, isOrdered=False,lower=0, upper=Inf(), hostClass=tr2pl)
factory.createProperty("weight", root=tr2pl["ownedProperty.value.element"],  isComposite=True, lower=1, upper=1, typ="INTEGER",\
    default=0, hostClass=tr2pl)
metamodel["ownedElement.value.size.value"].setValue(metamodel["ownedElement.value.size.value"].getValue()+1)

print metamodel["name.value"].getValue() + " created in the memory."

return metamodel
```

C

Define the Action Model for PTPN Model Simulation, Python Code

```
#PTPN Model Simulation Action Model
#Input:    The nodes in the found pattern.
#Output:   Action Model for simulation
from ArkM3.visitor.Factory import Factory
from ArkM3.DataType import * #IntegerType, RealType, BooleanType, StringType
from function.UniqueId import UniqueId

def rule_A_actioncode_model(root, x, seq):
    f = Factory()
    temp = f.createAction("rule_A", isImplemented = True, hostElement = root)
    temp.parameter = f.createSequenceValue()
    temp.parameter.add(x)
    temp.parameter.add(seq)

    temp.ownedStmt = f.createSequenceValue()
    temp.ownedStmt.element = [f.createExpressionStmt()]
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr = f.createSequenceValue()
    # FIXME: Here should use the Call operator.
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(rule_A_actioncode_model_imaptoseq\
        (root, temp.parameter.element[0]))
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(rule_A_actioncode_model_dmaptoseq\
        (root, temp.parameter.element[1]))
    return temp

def rule_A_actioncode_model_imaptoseq(root, seq):
    """
    print "                O  O       PetriNet.Place:a"
    print "                 \/      PetriNet.Transition2Place:c"
    print " PNpattern        ==     PetriNetTransition.:b"
    print " All the inputs to a need to satisfy place.token>p2t.weight"
    """
    f = Factory()
    temp = f.createAction("maptoseq", isImplemented = True, hostElement = root)
    temp.parameter = f.createSequenceValue()
    temp.parameter.add(seq)

    temp.ownedStmt = f.createSequenceValue()
    temp.ownedStmt.element = [f.createExpressionStmt()]
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr = f.createSequenceValue()

    a = f.createIdentifier("i")
    b = f.createIdentifier("j")
    tup = f.createTupleValue([a,b])

    act = rule_A_actioncode_model_inc(root, tup)

    expr_map = f.createMap()
    expr_map.add_child(tup)
    expr_map.add_child(temp.parameter.get(f.createIntegerValue(0)))
    expr_map.add_child(act)
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr_map)
    return temp

    #updatetoken = rule_A_actioncode_model_updatetokenvalue(root, seq, model)



def rule_A_actioncode_model_inc(root, tup):
    f = Factory()
```

```
    temp = f.createAction("rule_A_inc", isImplemented = True, hostElement = root)
    temp.parameter = f.createSequenceValue()
    temp.parameter.add(tup)

    temp.ownedStmt = f.createSequenceValue()
    temp.ownedStmt.element = [f.createExpressionStmt()]
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr = f.createSequenceValue()


    a = f.createIdentifier("i")
    i = f.createTupleLiteral([temp.parameter.get(f.createIntegerValue(0)), f.createIntegerValue(0)])
    expr1 = f.createAssignmentExpr(a, i)
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr1)

    b = f.createIdentifier("j")
    j = f.createTupleLiteral([temp.parameter.get(f.createIntegerValue(0)), f.createIntegerValue(1)])
    expr2 = f.createAssignmentExpr(b, j)
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr2)


    tokenref = f.createReference(["token", "value"])

    itokenvalue = f.createRead()
    itokenvalue.add_child(a)
    itokenvalue.add_child(tokenref)

    weightref = f.createReference(["weight", "value"])
    jweightvalue = f.createRead()
    jweightvalue.add_child(b)
    jweightvalue.add_child(weightref)

    rhs = f.createPlus()
    rhs.add_child(itokenvalue)
    rhs.add_child(jweightvalue)

    expr = f.createAssignmentExpr(itokenvalue, rhs)

    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr)

    return temp


def rule_A_actioncode_model_dmaptoseq(root, seq):
    """
    print " PNpattern        ==     PetriNetTransition.:b"
    print "                   \      PetriNet.Transition2Place:c"
    print "                   0       PetriNet.Place:a"
    print " All the inputs to a need to satisfy place.token>p2t.weight"
    """
    f = Factory()
    temp = f.createAction("maptoseq", isImplemented = True, hostElement = root)
    temp.parameter = f.createSequenceValue()
    temp.parameter.add(seq)

    temp.ownedStmt = f.createSequenceValue()
    temp.ownedStmt.element = [f.createExpressionStmt()]
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr = f.createSequenceValue()

    a = f.createIdentifier("i")
    b = f.createIdentifier("j")
    tup = f.createTupleValue([a,b])

    act = rule_A_actioncode_model_updatetokenvalue(root, tup)

    expr_map = f.createMap()
    expr_map.add_child(tup)
    expr_map.add_child(temp.parameter.get(f.createIntegerValue(0)))
    expr_map.add_child(act)
    temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr_map)
    return temp

    #updatetoken = rule_A_actioncode_model_updatetokenvalue(root, seq, model)


def rule_A_actioncode_model_updatetokenvalue(root, tup):
    f = Factory()
    temp = f.createAction("updatetokenvalue", isImplemented = True, hostElement = root)
    temp.parameter = f.createSequenceValue()
    temp.parameter.add(tup)
```

```
temp.ownedStmt = f.createSequenceValue()
temp.ownedStmt.element = [f.createExpressionStmt()]
temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr = f.createSequenceValue()

a = f.createIdentifier("i")
i = f.createTupleLiteral([temp.parameter.get(f.createIntegerValue(0)), f.createIntegerValue(0)])
expr1 = f.createAssignmentExpr(a, i)
temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr1)

b = f.createIdentifier("j")
j = f.createTupleLiteral([temp.parameter.get(f.createIntegerValue(0)), f.createIntegerValue(1)])
expr2 = f.createAssignmentExpr(b, j)
temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr2)

tokenref = f.createReference(["token", "value"])
itokenvalue = f.createRead()
itokenvalue.add_child(a)
itokenvalue.add_child(tokenref)

weightref = f.createReference(["weight", "value"])
jweightvalue = f.createRead()
jweightvalue.add_child(b)
jweightvalue.add_child(weightref)

rhs = f.createMinus()
rhs.add_child(itokenvalue)
rhs.add_child(jweightvalue)

expr = f.createAssignmentExpr(itokenvalue, rhs)
temp.ownedStmt.get(f.createIntegerValue(0)).ownedExpr.add(expr)

return temp
```