



Proceedings of the
8th International Workshop on Graph-Based Tools
(GraBaTs 2014)

Algebraic Specification to Realize the Diagram Predicate Framework

No \author defined!

14 pages

Algebraic Specification to Realize the Diagram Predicate Framework

No \author defined!

No \institute defined!

Abstract: Model Driven Engineering (MDE) is considered to be an effective way of improving the quality of software engineering, system design and development, and communication between the people concerned with these problems as MDE provides abstract representations of the knowledge and activities that govern a particular application domain. Different types of application domains require different domain specific languages (DSL) to increase productivity and simplify software design. DSLs are created specifically to solve problems for different application domains. A general but formal approach to develop domain specific modelling languages has a great demand in the software engineering field. Diagram Predicate Framework (DPF) is a formal approach for metamodelling that can be used to develop different modelling languages. In this article, we enhance DPF with rewriting rules and provide an executable environment for simulating DPF models.

Keywords: model driven engineering, model transformation, metamodelling, rewriting theory, workflow model

1 Introduction

Diagram Predicate Framework (DPF) provides a formal diagrammatic approach to metamodelling based on category theory and model transformation [Rut10]. In this approach models at any level are formalised as diagrammatic specifications which consist of underlying graphs and diagrammatic constraints. The graphs represent the structures of the models; the diagrammatic constraints (also known as predicate constraints) are added to the structure; the graphs and constraints together provide the modelling formalisms of DPF. A DPF metamodelling hierarchy consists of metamodels, models and instances of models. A metamodel specification gives us a modelling language, the specification of a model represents a software system and the instances of models represent possible states of a software system. DPF has been extended in [RMWL12, RWM12] for behavioural modelling. A workflow modelling language called DERF was developed and afterwards it was extended in [WRM12] to define (static) semantics for timed and compensable workflow models and defined the dynamic semantics of models by a transition system where the states are instances and transitions are applications of transformation rules.

In this article we present a new type of DPF construct named *rewriting predicate* that permits us to specify model transformation rules in order to derive inferred knowledge. We present a graphical way of specifying *rewriting predicates* allowing us to partially specify a model and derive a complete model by applying model transformation rules. This reduces the efforts required to explicitly specify the complete model of a system. So while designing a software model

with DPF, a designer gets different abstraction level, perspective from different granularity to understand, and ability to develop a rewriting system graphically using DPF.

We used algebraic specification to realize the static and dynamic semantics of DPF specifications. Although other executable systems may be used, to simulate the software model designed in DPF we propose the use of Maude system [BM06, CDE⁺07] which is a high-level programming/specification/modeling language based on rewriting logic theory.

In short, our main contribution in this article is as follows: 1) Extension of DPF with *rewriting predicates*; 2) Representation of DPF models in Maude. The paper is organized as follows: in section 2 we present *rewriting predicate*, in section 3 we provide the algebraic specification for DPF specifications, we present some related works in section 4, and in section ?? we conclude the paper with some future directions.

2 Rewriting Predicate

A diagrammatic specification is formally specified in [Rut10] as a tuple $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consisting of an underlying graph S together with a set of *atomic constraints* $C^{\mathfrak{S}}$. The predicate constraints are from a predefined *diagrammatic predicate signature* Σ and they are used to add constraints to S . We modify the definition of **Signature** from [Rut10] with Definition. 1 in order to incorporate *rewriting predicate* into Σ .

Definition 1 (Signature) A (diagrammatic predicate) signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma}, \pi^{\Sigma})$ consists of a collection of predicate symbols P^{Σ} with two multi-maps $\alpha^{\Sigma}, \pi^{\Sigma}$. The map α^{Σ} assigns graphs to predicate symbols $p \in P^{\Sigma}$ and $\alpha^{\Sigma}(p)$ is called the arity of the predicate symbol p . The map π^{Σ} assigns transformation rules to predicate symbols $p \in P^{\Sigma}$ and $\pi^{\Sigma}(p)$ is called the rewriting predicate of the predicate symbol p .

Remark 1 A predicate $p \in P^{\Sigma}$ is either a *predicate constraint* (denoted as $\text{dom}(\alpha^{\Sigma})$), iff $\alpha^{\Sigma}(p) \neq \emptyset$ or a *rewriting predicate* (denoted as $\text{dom}(\pi^{\Sigma})$) iff $\pi^{\Sigma}(p) \neq \emptyset$.

Table. 1, 2 shows a sample signature $\Sigma_2 = (P^{\Sigma_2}, \alpha^{\Sigma_2}, \pi^{\Sigma_2})$ where the first column of both tables show the names of the predicates; the second, third and fourth columns of Table. 1 show the arities of predicates, a possible visualization and the semantic interpretation respectively; the second and third columns of Table. 2 show the transformation rules of predicates and a possible visualization respectively. For readability reasons, the semantic interpretations of *predicate constraints* are presented in a set-theoretical indexed manner in Table. 1, although the actual semantics of each predicate constraint $p \in P^{\Sigma}$ of a sample signature Σ is defined in [Rut10] by a set $\llbracket p \rrbracket^{\Sigma}$ of graph homomorphisms $\iota : O \rightarrow \alpha^{\Sigma}(p)$ called valid instances of p , where O may vary over all graphs. For example, the valid instances of *[inverse] predicate constraint* of signature Σ_2 (see Table. 1) are defined as follows. All valid instances of $\llbracket \text{inverse} \rrbracket^{\Sigma_2}$ that provides the semantics of *[inverse]* are provided below:

Table 1: Predicate constraints of a sample signature Σ_2

p	$\alpha^{\Sigma_2}(p)$	Proposed Vis.	Semantic Interpretation
[mult(n,m)]	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[n..m]{f} \boxed{Y}$	$\forall x \in X : m \leq f(x) \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[irreflexive]	$1 \xrightarrow{f} 1$	$\boxed{X} \xrightarrow{f} \boxed{X}$ [irr]	$\forall x \in X : x \notin f(x)$
[injective]	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow{[inj]} \boxed{Y}$	$\forall x, x' \in X : f(x) = f(x') \text{ implies } x = x'$
[surjective]	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow{[sur]} \boxed{Y}$	$\forall y \in Y, \exists x \in X, f(x) = y$
[inverse]	$1 \xrightleftharpoons{f,g} 2$	$\boxed{X} \xrightleftharpoons{[inv]} \boxed{Y}$	$\forall x \in X, \forall y \in Y : y \in f(x) \text{ iff } x \in g(y)$
[image-inclusion]	$1 \xrightleftharpoons{f,g} 2$	$\boxed{X} \xrightleftharpoons{[\subseteq]} \boxed{Y}$	$\forall x \in X : f(x) \subseteq g(x)$

$$\begin{aligned}
l_1 : \left(a \xrightleftharpoons{f,g} b \right) &\longrightarrow \left(1 \xrightleftharpoons{f,g} 2 \right) \\
l_2 : \left(\begin{array}{cc} a & b \end{array} \right) &\longrightarrow \left(1 \xrightleftharpoons{f,g} 2 \right) \\
l_3 : \left(\begin{array}{c} \\ b \end{array} \right) &\longrightarrow \left(1 \xrightleftharpoons{f,g} 2 \right) \\
l_4 : \left(\begin{array}{c} a \\ \\ \end{array} \right) &\longrightarrow \left(1 \xrightleftharpoons{f,g} 2 \right) \\
l_5 : \left(\begin{array}{c} \\ \\ \end{array} \right) &\longrightarrow \left(1 \xrightleftharpoons{f,g} 2 \right)
\end{aligned}$$

On the other hand, the semantics of *rewriting predicates* are defined by coupled transformation [SLK11] rules with negative application conditions as in Table. 2.

Table 2: Rewriting predicates of a sample signature Σ_2

p	$\pi^{\Sigma_2}(p)$		Proposed Vis.
	$(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) \xleftarrow{n} (\mathfrak{L}_0 \rightarrow \mathfrak{L}_1)$	$(\mathfrak{R}_0 \rightarrow \mathfrak{R}_1)$	
[opposite]	$\boxed{x:X} \xrightleftharpoons{f/g} \boxed{y:Y}$	$\boxed{x:X} \xrightleftharpoons{f} \boxed{y:Y}$	$\boxed{X} \xrightleftharpoons{[ops]} \boxed{Y}$
	$\boxed{x:X} \xrightleftharpoons{f/g} \boxed{y:Y}$	$\boxed{x:X} \xrightleftharpoons{f} \boxed{y:Y}$	
[composition]	$\boxed{x:X} \xrightarrow{f} \boxed{y:Y} \xrightarrow{g} \boxed{z:Z}$ $\boxed{x:X} \xrightarrow{h} \boxed{z:Z}$	$\boxed{x:X} \xrightarrow{f} \boxed{y:Y} \xrightarrow{g} \boxed{z:Z}$ $\boxed{x:X} \xrightarrow{h} \boxed{z:Z}$	$\boxed{X} \xrightarrow{f} \boxed{Y} \xrightarrow{g} \boxed{Z}$ $\boxed{X} \xrightarrow{h} \boxed{Z}$ [comp]

Definition 2 (Semantics of Rewriting Predicates) The semantics of a rewriting predicate $p \in$

$P^{\Sigma_{i+1}}$ is given by a set of coupled transformation rules where each transformation rule $r : (\mathcal{L}_i \rightarrow \mathcal{L}_{i+1}) \leftarrow (\mathcal{R}_i \rightarrow \mathcal{R}_{i+1}) \rightarrow (\mathcal{N}_i \rightarrow \mathcal{N}_{i+1})$ has a negative application condition and there exists a specification morphism $n : (\mathcal{L}_i \rightarrow \mathcal{L}_{i+1}) \rightarrow (\mathcal{N}_i \rightarrow \mathcal{N}_{i+1})$.

Notice that the matching patterns $(\mathcal{L}_i \rightarrow \mathcal{L}_{i+1})$ and negative application conditions $(\mathcal{N}_i \rightarrow \mathcal{N}_{i+1})$ are represented in the same diagram in Table. 2 to make it more readable. Negative application conditions are striked through in the diagram to represent that they must not exist while matching. Now it is worthwhile to make a comparison between these two types of predicates. *Predicate constraints* do not derive any change to an instance of a specification whereas *rewriting predicates* may change the structure of an instance. By defining *predicate constraints* we can specify some structural constraints and can decide if an instance is a valid instance of a specification. In order to be consistent we need to modify the definition of **Atomic Constraint** and **Specification** from [Rut10] and include a new definition for **Rewriting Rule** as follows:

Definition 3 (Atomic Constraint) Given a signature $\Sigma = (P^\Sigma, \alpha^\Sigma, \pi^\Sigma)$, an atomic constraint (p, δ) added to a graph S is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^\Sigma(p) \rightarrow S$ where $p \in P^\Sigma$ is a predicate.

Definition 4 (Rewriting Rule) Given a signature $\Sigma = (P^\Sigma, \alpha^\Sigma, \pi^\Sigma)$, a rewriting rule (r, δ) attached to a graph S is given by a predicate symbol r and a graph homomorphism $\delta : \pi^\Sigma(r) \rightarrow ((I, \iota) \rightarrow S)$ where $r \in P^\Sigma$ is a predicate.

Definition 5 (Specification) Given a signature $\Sigma = (P^\Sigma, \alpha^\Sigma, \pi^\Sigma)$, a (diagrammatic) specification $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma, R^\mathfrak{S} : \Sigma)$ is given by a graph S , a set $C^\mathfrak{S}$ of atomic constraints (p, δ) on S with $p \in \text{dom}(\alpha^\Sigma)$, and a set $R^\mathfrak{S}$ of rewriting rules (r, δ) on S with $r \in \text{dom}(\pi^\Sigma)$.

Lets take the following requirements of a healthcare domain to develop an entity diagram using DPF:

1. An employee (e.g., nurse, doctor) must work for a department.
2. A department may have zero or more employees.
3. An ward must be under a department.
4. An employee who is involved in a ward, must work in the controlling department.
5. Patients systolic and diastolic blood pressure records are natural numbers.

Fig. 1(a) shows an entity specification \mathfrak{S}_1 that we developed from the above mentioned requirements. In this specification we used the signature Σ_2 from Table. 1 and 2 to define the set $C^{\mathfrak{S}_1}$ of atomic constraints (see Table. 3) and a set $R^{\mathfrak{S}_1}$ of rewriting rules (see Table. 4). Requirement 1 is encoded in \mathfrak{S}_1 by the [surjective] predicate constraint over the morphism $depEmps$; the fourth requirement “an employee who is involved in a ward, must work in the controlling department” is encoded in \mathfrak{S}_1 by a rewriting predicate [composition] and a predicate constraint [image – inclusion] over morphisms $depEmps$, $wardDeps$, $wardEmps$ and $wardEmps'$ where $wardEmps := wardDeps; depEmps$ (the composition of morphisms $wardDeps$ and $depEmps$).

Fig. 1(a) shows a possible instance (I, ι) of \mathfrak{S}_1 . Even though I is typed by S , it is not a valid instance of \mathfrak{S}_1 since it does not satisfy the predicate constraints [surjective], [inverse] and [image – inclusion]. We explain this by a pullback operation in Fig. 2. Since $\iota^* \notin \llbracket \text{inverse} \rrbracket^{\Sigma_2}$,

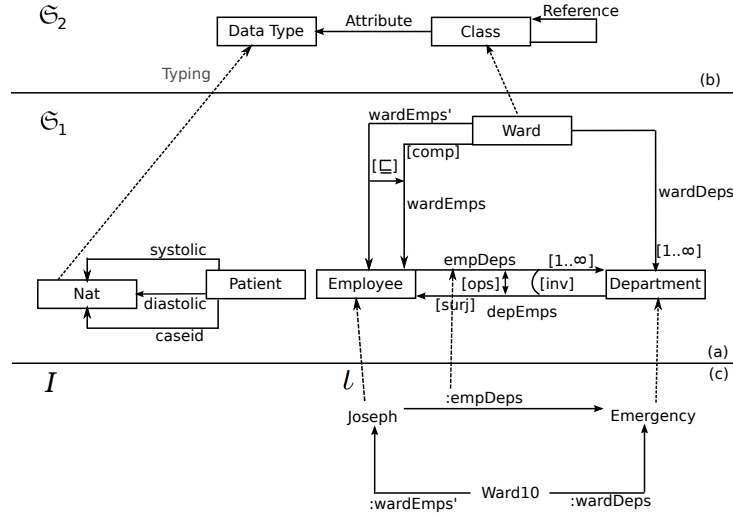


Figure 1: a) Entity specification $\mathfrak{S}_1 = (S, C^{\mathfrak{S}_1} : \Sigma_2, R^{\mathfrak{S}_1} : \Sigma_2)$, b) its meta model \mathfrak{S}_2 , and c) an instance (I, ι) of \mathfrak{S}_1

I does not satisfy *[inverse]*. Similarly we can explain why I does not satisfy *[surjective]* and *[image – inclusion]*.

Until at this point we have not considered the execution of *rewriting rules* over (I, ι) . By applying *rewriting rules* from Table. 4 over (I, ι) we may derive new information. An instance of a specification where *rewriting rules* are applicable is called a partial instance. The *rewriting predicate [opposite]* is used to derive morphism $y \xrightarrow{g} x$ (if it does not exist) from the existence of a morphism $x \xrightarrow{f} y$ or vice versa; *[composition]* is used to derive morphism $z \xrightarrow{h} x$ (if it does not exist) from the existence of morphisms $z \xrightarrow{g} y$ and $y \xrightarrow{f} x$; Once we apply the *rewriting rules* for *[opposite]*, *[composition]* over (I, ι) we get a complete instance of specification \mathfrak{S}_1 as shown in Fig. 3. We call it a complete instance of \mathfrak{S}_1 as no more *rewriting predicates* are further applicable on (I, ι) . This complete instance is a valid instance of \mathfrak{S}_1 as it satisfies all atomic constraints. Now it is easy to understand the motivation of using *rewriting predicates* in DPF as it permits us to specify partial instance of a specification. We may utilize the rewriting capability of *rewriting predicates* to derive a complete instance from a partially specified instance.

Definition 6 (Application of Rewriting Rules) Let $(\Sigma_1 \triangleright S_1, \mathfrak{S}_1, \Sigma_2)$ be a modelling formalism where specification $\mathfrak{S}_1 = (S_1, C^{\mathfrak{S}_1} : \Sigma_2, R^{\mathfrak{S}_1} : \Sigma_2)$ has a set $R^{\mathfrak{S}_1}$ of rewriting rules (r, δ) on S_1 . Each rewriting rule (r, δ) is associated with a set of coupled transformation rules $\pi^{\Sigma_2}(r)$. Each transformation rule $t \in \pi^{\Sigma_2}(r)$ has a negative application condition. A rule $t : (\mathcal{L}_0 \rightarrow \mathcal{L}_1) \leftarrow (\mathfrak{R}_0 \rightarrow \mathfrak{R}_1) \rightarrow (\mathfrak{N}_0 \rightarrow \mathfrak{N}_1)$ is applicable over $\mathfrak{S}_0 \triangleright S_1$ if there is a match $m : \delta(\mathcal{L}_0 \rightarrow \mathcal{L}_1) \rightarrow (\mathfrak{S}_0 \rightarrow \mathfrak{S}_1)$ and there does not exist any match to the negative application condition, $q : \delta(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) \rightarrow (\mathfrak{S}_0 \rightarrow \mathfrak{S}_1)$ such that $m = n; q$.

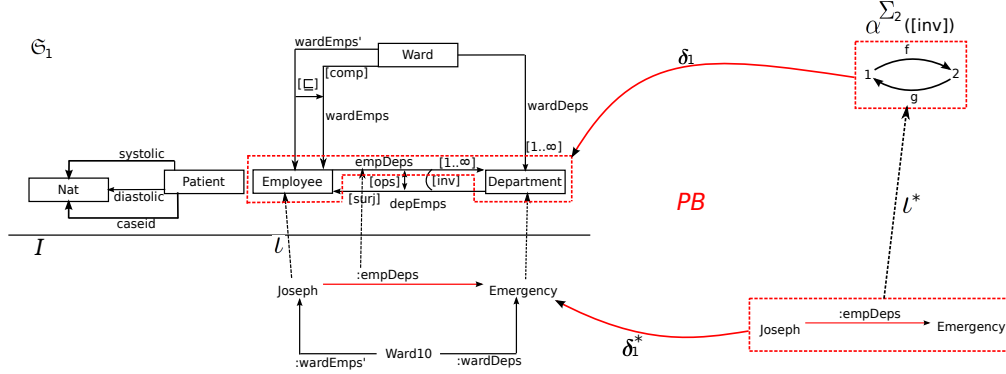

 Figure 2: Pullback $\alpha^{\Sigma_2}([inv]) \xleftarrow{l^*} O^* \xrightarrow{\delta_1^*} I$ of $\alpha^{\Sigma_2}([inv]) \xrightarrow{\delta_1} S \xleftarrow{l} I$

 Table 3: The set of atomic constraints $C^{\mathfrak{S}_1}$ of \mathfrak{S}_1

(p, δ)	$\alpha^{\Sigma_2}(p)$	$\delta(\alpha^{\Sigma_2}(p))$
$([mult(1, \infty)], \delta_1)$	$1 \xrightarrow{f} 2$	$\text{Employee} \xrightarrow{\text{empDeps}} \text{Department}$
$([mult(1, \infty)], \delta_2)$	$1 \xrightarrow{f} 2$	$\text{Ward} \xrightarrow{\text{wardDeps}} \text{Department}$
$([surjective], \delta_3)$	$1 \xrightarrow{f} 2$	$\text{Department} \xrightarrow{\text{depEmps}} \text{Employee}$
$([inverse], \delta_4)$	$1 \begin{smallmatrix} \xrightarrow{f} \\ \xleftarrow{g} \end{smallmatrix} 2$	$\text{Employee} \begin{smallmatrix} \xrightarrow{\text{depEmps}} \\ \xleftarrow{\text{empDeps}} \end{smallmatrix} \text{Department}$
$([image - inclusion], \delta_5)$	$1 \begin{smallmatrix} \xrightarrow{f} \\ \xleftarrow{g} \end{smallmatrix} 2$	$\text{Ward} \begin{smallmatrix} \xrightarrow{\text{wardEmps}} \\ \xleftarrow{\text{wardEmps}'} \end{smallmatrix} \text{Employee}$

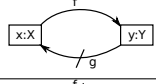
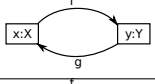
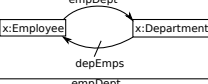
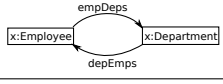
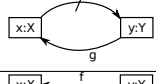
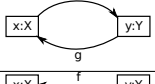
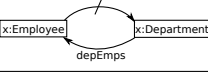
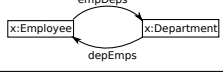
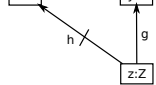
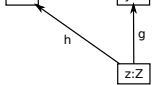
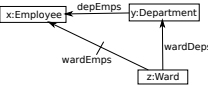
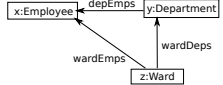
$$\begin{array}{ccccc}
 \delta(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) & \xleftarrow{n} & \delta(\mathfrak{L}_0 \rightarrow \mathfrak{L}_1) & \xleftarrow{m} & \delta(\mathfrak{K}_0 \rightarrow \mathfrak{K}_1) & \xrightarrow{m'} & \delta(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) \\
 & \searrow q & \downarrow m & & \downarrow m' & & \downarrow m^* \\
 & & (\mathfrak{S}_0 \rightarrow \mathfrak{S}_1) & \xleftarrow{q} & (\mathfrak{S}_0' \rightarrow \mathfrak{S}_1') & \xrightarrow{m^*} & (\mathfrak{S}_0^* \rightarrow \mathfrak{S}_1^*)
 \end{array}$$

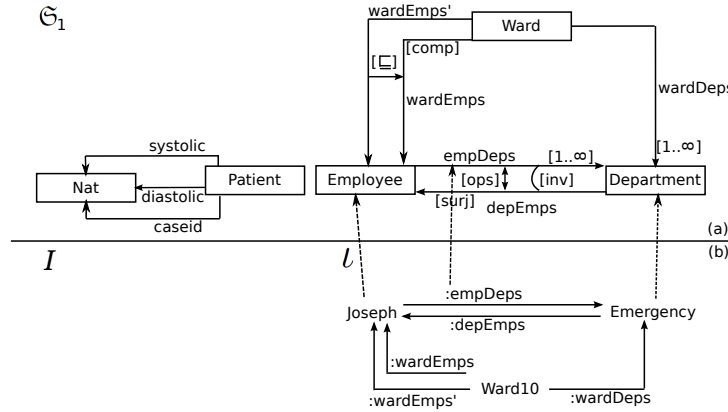
Definition 7 (Partial and Complete Instance) Let $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma, R^{\mathfrak{S}} : \Sigma)$ be a specification with a graph S , a set $C^{\mathfrak{S}}$ of atomic constraints (p, δ) on S with $p \in \text{dom}(\alpha^{\Sigma})$, and a set $R^{\mathfrak{S}}$ of rewriting rules (r, δ) on S with $r \in \text{dom}(\pi^{\Sigma})$. An instance (I, ι) is a *partial instance* of \mathfrak{S} if there exist some rewriting rules $(r, \delta) \in R^{\mathfrak{S}}$ those are applicable over (I, ι) . An instance (I, ι) is a *complete instance* of \mathfrak{S} if there does not exist any rewriting rule $(r, \delta) \in R^{\mathfrak{S}}$ that is applicable over (I, ι) .

Remark 2 A complete instance (I, ι) is a *valid instance* of specification \mathfrak{S} if it does not violate any constraint from $C^{\mathfrak{S}}$. We denote it as $(I, \iota) \blacktriangleright \mathfrak{S}$

It should be clear from above text that the application of *rewriting rules* need to be applied over

Table 4: The set of rewriting rules $R^{\mathfrak{S}_1}$ of \mathfrak{S}_1

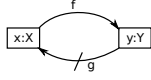
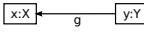
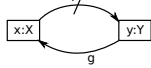
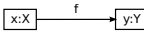
(r, δ)	$\pi^{\Sigma_2}(r)$		$\delta(\pi^{\Sigma_2}(r))$	
	$(\mathfrak{V}_0 \rightarrow \mathfrak{V}_1) \xleftarrow{n} (\mathfrak{L}_0 \rightarrow \mathfrak{L}_1)$	$(\mathfrak{R}_0 \rightarrow \mathfrak{R}_1)$	$\delta(\mathfrak{V}_0 \rightarrow \mathfrak{V}_1) \xleftarrow{n} \delta(\mathfrak{L}_0 \rightarrow \mathfrak{L}_1)$	$\delta(\mathfrak{R}_0 \rightarrow \mathfrak{R}_1)$
$([opposite], \delta_6)$				
				
$([composition], \delta_7)$				


Figure 3: a) Entity specification $\mathfrak{S}_1 = (\mathcal{S}, C^{\mathfrak{S}_1} : \Sigma_2, R^{\mathfrak{S}_1} : \Sigma_2)$ and b) a complete instance (I, t) of \mathfrak{S}_1

an instance (I, t) before checking *atomic constraints*. By utilizing the *rewriting predicates* we may develop a rewriting system specification with sufficient reasoning capacity. The formulation of the *rewriting predicates* are very expressive since we used double push out approach for the transformation rules and it is possible to mention some negative application conditions. One issue with such expressive rules is that the rewriting systems may suffer non-termination. Table. 5 shows two *rewriting predicates* p_1, p_2 of a signature Σ_2 that can possibly make a specification non-terminating. The application of any of the transformation rules from p_1, p_2 make the other one active; therefore the application of the rewriting rules over an instance of a specification would iterate forever.

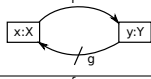
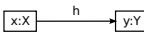

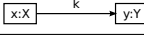
Another issue with this expressive rules is that the DPF system does not restrict the user from writing non-deterministic *rewriting predicates*, and therefore the order of application of *rewriting rules* over an instance (I, t) may provide different instances. In literature this issue is attributed as confluent property of a reduction system [BN98]. Table. 6 shows two *rewriting predicates* q_1, q_2 of a sample signature Σ_2 to demonstrate the non-confluent nature of DPF rewriting system.

Table 5: Non terminating rewriting predicates of a sample signature Σ_2

p	$\pi^{\Sigma_2}(p)$	
	$(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) \xleftarrow{n} (\mathfrak{L}_0 \rightarrow \mathfrak{L}_1)$	$(\mathfrak{R}_0 \rightarrow \mathfrak{R}_1)$
$[p_1]$		
$[p_2]$		

The application of the *rewriting predicates* give us different results even though the patterns and negative application conditions are same. So the application of these predicates over an instance would give us different end results. In these scenarios, we may filter out some invalid instances those do not satisfy the set of atomic constraints. After sorting out all complete and valid instances, the user might be provided with different options to choose the instance he is interested to work with.

Table 6: Non confluent rewriting predicates of a sample signature Σ_2

p	$\pi^{\Sigma_2}(p)$	
	$(\mathfrak{N}_0 \rightarrow \mathfrak{N}_1) \xleftarrow{n} (\mathfrak{L}_0 \rightarrow \mathfrak{L}_1)$	$(\mathfrak{R}_0 \rightarrow \mathfrak{R}_1)$
$[p_1]$		
$[p_2]$		

3 Algebraic Specification for Metamodel Integration

One of our intension to use algebraic specification for DPF metamodeling is to develop an execution environment for DPF framework where one could design and simulate a software model with multiple metamodels. Fig. 4 shows a proposed system architecture for DPF simulation tool where maude tool has been used as an engine to perform 1) Type checking, 2) Conformance checking, and 3) Rewriting / derivation. During the design phase, the tool may be used to provide feedback to the designer if the model he is working on satisfies its metamodel or not. We intend to use a user-friendly user interface (UI) to support better user experience for the developer to simulate a model instance. During the simulation phase, maude tool will be used to produce the instances of a software model.

Since DPF is based on metamodeling, a natural choice would be to use the META-LEVEL programming capability of Maude specification. A detailed description about maude metalevel programming and metalanguage applications may be found from Chapters 14 and 17 of [CDE⁺07].

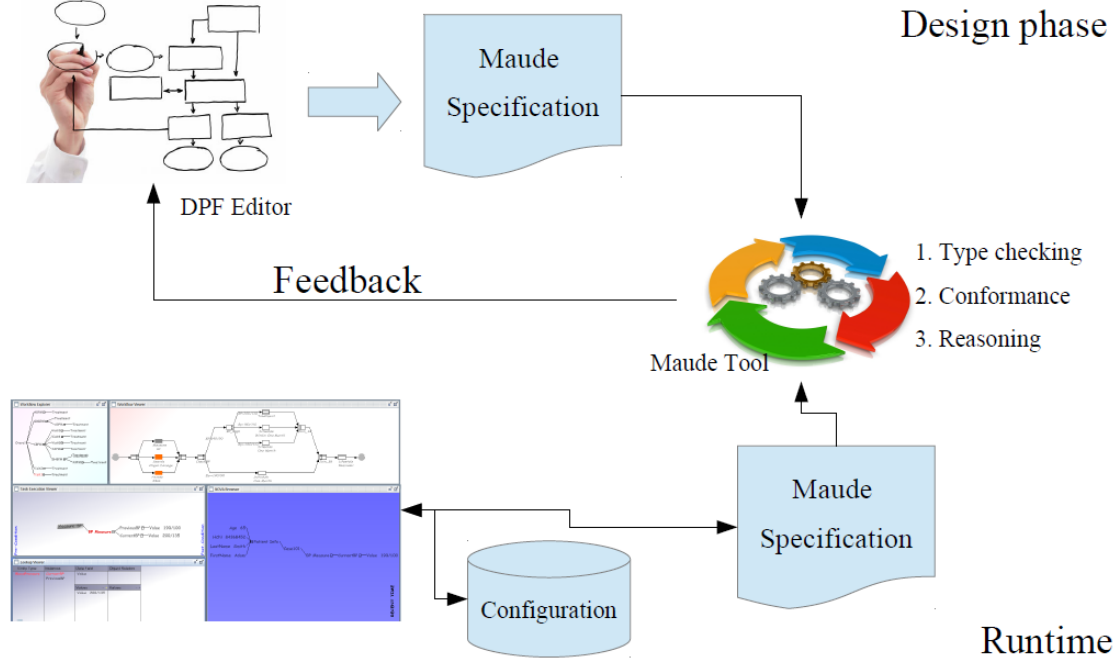


Figure 4: System architecture for DPF simulation tool

Maude metalevel programming is based on a “reflective tower” with an arbitrary number of levels of reflection. We can represent in a rewrite theory U , any finitely presented rewrite theory R (including U itself) as a term \bar{R} , any terms t, t' in R as terms \bar{t}, \bar{t}' , and any pair (R, t) as a term $\langle \bar{R}, \bar{t} \rangle$.

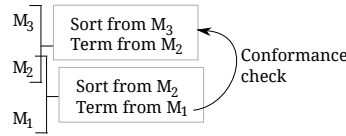
$$R \vdash t \rightarrow t' \Leftrightarrow U \vdash \langle \bar{R}, \bar{t} \rangle \rightarrow \langle \bar{R}, \bar{t}' \rangle \Leftrightarrow U \vdash \langle \bar{U}, \langle \bar{R}, \bar{t} \rangle \rangle \rightarrow \langle \bar{U}, \langle \bar{R}, \bar{t}' \rangle \rangle \dots$$

In this chain of equivalences the first computation takes place at level 0, the second computation at level 1, and so on. This is a barrier to use maude metalevel programming for DPF as the computation of a single step requires considerable computational cost that involves many rewriting steps one level up. For this reason it was suggested for maude programs to lower the number of levels of reflective computation as much as possible, so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary [CDE⁺07]. But the semantic of DPF metamodel specifications are provided in the opposite way. Once a specification \mathfrak{S}_i is defined in DPF, the computation regarding type checking, conformance checking and rewriting of the instance \mathfrak{S}_{i+1} of the specification \mathfrak{S}_i does not involve any computation regarding the type checking, conformance checking or rewriting of \mathfrak{S}_i or any of its metamodel \mathfrak{S}_{i-1} , \mathfrak{S}_{i-2} , etc., anymore.

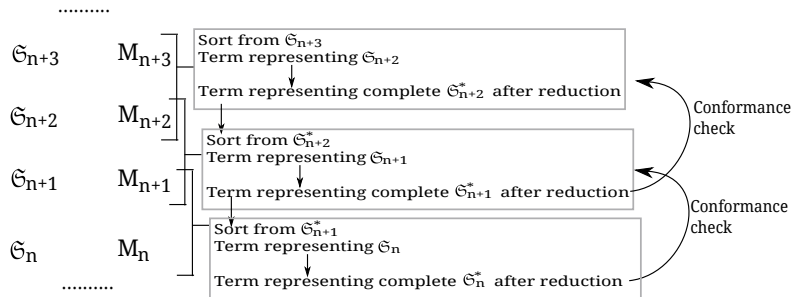
Another issue regarding the use of maude metalevel programming is the complex sort infrastructure provided by the module META-TERM. In order to write a rewriting rule at a higher level, requires the use of metarepresentation of terms. The metarepresentation of terms, when they are iterated, becomes very complex; for example, the metarepresentation of a natural num-

ber such as, e.g., 42 is $'s_{-}^{42}[0.Zero]$. If we wish to make a reusable DPF specification, this would become a big issue as the rewriting theories metarepresented in one signature may not match in another metalevel hierarchy. In general, for DPF, while we are specifying signatures for a higher level specification, we should not practice using a concept/name from a specification at lower metalevel.

An algebraic semantics for MOF was presented in [BM10] which is very relevant with our situation. In that paper the authors proposed an algebraic, mathematical semantics for MOF in membership equational logic (MEL). The algebraic semantics that they proposed, exploits the reflective features of both MOF and MEL and they used a modular and stepwise approach in the definition of the semantic function. An abstraction of the reflective bootstrapping strategy to define the semantic function is shown here:



A straightforward translation of DPF specifications to maude using this approach would not work since the specifications in DPF have dynamic semantics; so the *sort* declaration for any meta level n requires parsing of the reduced term at level $n + 1$. This is pictorially represented in the following diagram:



Another complexity is to deal with the integrated specification since it requires to perform a disjoint union operation over the underlying graphs of two specifications and the pullback operation after reduction over the term representing the integrated specification.

We present an object oriented stepwise approach in maude specification where each object represent a specification consisting of typed graphs and their association with predicates. Predicate constraints, rewriting predicates, and type checking using graph homomorphism are encoded as conditional rewriting rules. We control the execution of the rewriting rules by an external program written in Java. Terms of sort *Object* are defined by means of the constructor in maude specification:

$$\text{op } < _ : _ > : \text{Oid Cid AttributeSet} \rightarrow \text{Object [ctor object]} .$$

Part of *sorts*, *subsorts* and *operator* declarations of maude specification related to the *AttributeSet* is given below:

```

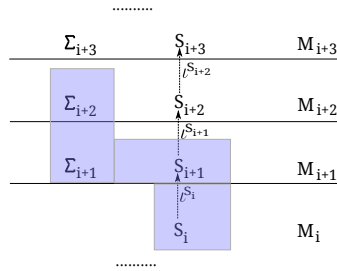
sorts Attribute AttributeSet Atom SigAtom DpfGraph SignatureMaps SIGID .
subsort Atom < DpfGraph .
subsort SigAtom < SignatureMaps .

op _,_ : AttributeSet AttributeSet -> AttributeSet
  [ctor assoc comm id: none] .
op relations :_ : DpfGraph -> Attribute [ctor gather (&)] .
op sigMaps :_ : SignatureMaps -> Attribute [ctor gather (&)] .

op rel : String String String String -> Atom .
op rel : String String String Nat -> Atom .
...
op sigmap : SIGID String String -> SigAtom .

```

Our stepwise translation approach is depicted in the following diagram. Consider there are n number of meta levels: M_0, M_1, \dots, M_{n-1} , in every step we translate a metamodeling formalism $\Sigma_{i+1}, \mathfrak{S}_{i+1}, \Sigma_{i+2}$ and a specification \mathfrak{S}_i where $\mathfrak{S}_{i+1} \triangleright \mathfrak{S}_{i+2}$, and \mathfrak{S}_i is a partial instance of \mathfrak{S}_{i+1} . After applying the rewriting rules from $R^{\mathfrak{S}_{i+1}}$ over \mathfrak{S}_i we check, if the derived specification is valid or not using the set $C^{\mathfrak{S}_{i+1}}$ of predicate constraints attached to \mathfrak{S}_{i+1} . If the derived specification is found to be a valid instance of \mathfrak{S}_{i+1} , then used as a meta model in the next step of translation.



Terms using the *rel* and *sigmap* operators presented in the above maude specification are used to hold a typed graph and a signature map respectively using multimaps. The arguments of a *rel* term represents a *name*, *type*, *domain* and *range* respectively. Here *name* is either referring to the name of a vertex or a morphism depending on the *type*. The arguments of a *sigmap* term represents an *Id*, *name* of a vertex or morphism, and *predicate name* respectively. A sample maude specification is provided below that represents the specification \mathfrak{S}_1 and its instance I from Fig. 1. The subterm *rel*("G0", "TypeOf", "Employee", "Class") represents that *Employee* is a vertex of type *Class*, and the subterm *rel*(":Typeof", "TypeOf", "Joseph", "Employee") represents that *Joseph* is a vertex of type *Employee*. Here *G0* denotes that *Employee* vertex will be treated as a metalevel node. The subterm *rel*("empDeps", "G1", "Employee", "Department") represents that *empDeps* is a morphism where the source is *Employee* and target is *Department*. The subterm *rel*(":empDeps", "empDeps", "Joseph", "Employee") represents an instance of the morphism *empDeps* from source *Joseph* to *Emergency*. Each *sigmap* subterm represents an association from the arity of predicates to the shape graph.

```

op initState : -> Configuration .
eq initState =
< DpfInstance : DPF |
  relations :
    rel("G0", "TypeOf", "Employee", "Class")
    rel("G0", "TypeOf", "Department", "Class")
    rel("G0", "TypeOf", "Ward", "Class")
    ...
    rel("empDeps", "G1", "Employee", "Department")
    rel("depEmps", "G1", "Department", "Employee")

```

```

....
rel(":TypeOf", "TypeOf", "Joseph", "Employee")
rel(":TypeOf", "TypeOf", "Emergency", "Department")
rel(":TypeOf", "TypeOf", "Ward10", "Ward")
rel(":empDeps", "empDeps", "Joseph", "Emergency")
,
sigMaps :
sigmap(Sig3, "empDeps", "[ops]" ) sigmap(Sig3, "depEmps", "[ops]" )
sigmap(Sig4, "wardEmps'", "[IMG-INC]-S") sigmap(Sig4, "wardEmps", "[IMG-INC]-T")
....
> .

```

We translated *rewriting predicates* to *conditional rewrite rules* following maude specification syntax. The translation of the *[opposite]* predicate from Table. 2 is provided below. The matching pattern checks for an instance (*morph*) of a morphism (*morphType*) that is associated with the predicate *[ops]*. The negative application condition is specified as a *if* statement: the statement is satisfied if an instance of opposite morphism (*opMorph*) does not exists with the source 'Z' and target 'X'. The replacement pattern is specified as a term that creates an instance of an opposite morphism (*opMorph*) with source 'Z' and target 'X'.

```

vars X Z PropName PropType OpProp : String .
var SigId : SIGID .
var G : DpfGraph .
var SG : SignatureMaps .

crl [ops] : < DpfInstance : DPF | relations : rel(morph, morphType, X, Z) G ,
    sigMaps : sigmap(SigId, morphType, "[ops]") sigmap(SigId, opMorph , "[ops]") SG >
=> < DpfInstance : DPF | relations : rel(morph, morphType, X, Z) rel(":"prop", opMorph, Z, X) G ,
    sigMaps : sigmap(SigId, morphType, "ops") sigmap(SigId, opMorph , "ops") SG >
if not relation-exists(opMorph, Z, X, G) .

```

We developed a Java program to externally follow a strategy in order to execute maude programs in different phases. During the first phase it prepares a maude system module where it loads all conditional rules related to *rewriting predicates*, the initial configuration of a DPF instance, and execute the following *search* command to get a configuration where no more rules are applicable. We get all possible complete instances by executing this command.

```
search initState =>! Ob:Object .
```

In the next phase, for each complete instance, the Java program prepares a maude system module where it loads the conditional rules related to *predicate constraints* and *graph homomorphisms* for type checking. The following code fragment shows the *crl* rule that verifies if a specification satisfies the *[image – inclusion]* predicate constraint from Table. 1.

```

crl [IMG-INC] : < DpfInstance : DPF | relations : rel(R1, morphType1, X, Y) G ,
    sigMaps : sigmap(SigId, morphType1, "[IMG-INC]-S") sigmap(SigId, morphType2, "[IMG-INC]-T") SG >
=> error (RULE-IMG-INC, rel(R1, morphType2, X,Y) )
if not relation-exists(morphType2, X, Y, G) .

```

The following maude specification checks graph homomorphism for DPF instances if they are type correct.

```

vars X Y Z R1 R2 W Z : String .
..
crl [Phi0] : < DpfInstance : DPF | relations : rel(R1, "TypeOf", X, Y) G , sigMaps : SG >
=> error (RULE-PHI0, rel(R1, "TypeOf", X, Y) )
if (R1 /= "G0") /\ (not type-exists("TypeOf", Y, G) ) .

crl [Phi1] : < DpfInstance : DPF | relations : rel(R1, R2, W, Z) rel(R2, "G1", X, Y) G , sigMaps : SG >
=> error (RULE-PHI1, rel(R1, R2, W, Z) rel(R2, "G1", X, Y) )
if (R2 /= "G1") /\ ((not supertype-exists(W, X, G)) or (not supertype-exists(Z, Y, G))) .

```

We use the following *search* command to verify if a complete DPF instance is a valid instance or not:

```
search finalState =>! < DpfInstance : DPF | A:AttributeSet > .
```

4 Related Work

Maude is a formal declarative programming language based on the mathematical theory of rewriting logic; it is a state-of-the-art formal tool in the fields of algebraic specification [vL90]. Custom data types, operators and their syntax may be defined algebraically by equations. Maude supports Membership Equational Logic (MEL), Object-Oriented programming including multiple inheritance and asynchronous communication through message passing. The dynamic behavior of a system is defined by rewrite rules which can be conditional/unconditional. Although Maude specification is powerful enough to develop the syntax and semantics of a new domain specific language, it is very difficult to use in practice because the language is not graphical and it has a steep learning curve.

Alloy is a structural modelling language for describing structural properties [Jac02]. The language offers graphical object models based on set-based formula syntax which is capable of expressing complex structural constraints and behaviour yet amenable to a fully automatic semantic analysis through the use of a boolean satisfiability problem (SAT) solver. For a given logical formula/metamodel, Alloy attempts to find a model which satisfies the formula/metamodel. The Alloy analyser attempts to find counterexamples within a limited scope which violates the constraints of the system. In our situation our problem is formulated in different way as we apply *rewriting rules* over a partial instance of a specification in order to produce a complete instance and afterwards validate if the complete instance conforms to its meta model by type checking and satisfying *atomic constraints*.

A relevant research topic to design and develop reasoning systems is OWL Ontology language. The language is designed to support the Semantic Web. The semantic of OWL is based on description logic [BCM⁺10] which is a decidable fragment of first-order predicate logic. The OWL language focus on describing classes and roles, and have a set-theoretic semantics to capture knowledge about some domain of interest. Since OWL Ontology keeps only TBox (Terminology) and ABox (Assertion) information, it does not properly support metamodeling.

Currently we are investigating how an integrated system model may be translated to an algebraic specification to realize the static and dynamic semantics of DPF specifications and the integration of multiple metamodels. In future we will investigate on strategies that should be followed to make DPF rewriting process confluent.

Bibliography

- [BCM⁺10] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2010.
<http://books.google.no/books?id=c35WRQAACAAJ>

- [BM06] R. Bruni, J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *Theor. Comput. Sci.* 360(1):386–414, Aug. 2006.
- [BM10] A. Boronat, J. Meseguer. An algebraic semantics for MOF. *Formal Asp. Comput.* 22(3-4):269–296, 2010.
- [BN98] F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.
- [CDE⁺07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (eds.). *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science 4350. Springer, 2007.
- [Jac02] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11(2):256–290, Apr. 2002.
[doi:10.1145/505145.505149](https://doi.org/10.1145/505145.505149)
<http://doi.acm.org/10.1145/505145.505149>
- [vL90] J. van Leeuwen (ed.). *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier and MIT Press, 1990.
- [RMWL12] A. Rutle, W. MacCaull, H. Wang, Y. Lamo. A Metamodelling Approach to Behavioural Modelling. In *Proceedings of BM-FA 2012: 4th Workshop on Behavioural Modelling: Foundations and Applications*. Pp. 5:1–5:10. ACM, 2012.
- [Rut10] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.
- [RWM12] A. Rutle, H. Wang, W. MacCaull. A Formal Diagrammatic Approach to Compensable Workflow Modelling. In Weber and Perseil (eds.), *FHIES*. Lecture Notes in Computer Science 7789, pp. 194–212. Springer, 2012.
- [SLK11] C. Schulz, M. Löwe, H. König. A Categorical Framework for the Transformation of Object-oriented Systems: Models and Data. *J. Symb. Comput.* 46(3):316–337, Mar. 2011.
- [WRM12] H. Wang, A. Rutle, W. MacCaull. A Formal Diagrammatic Approach to Timed Workflow Modelling. In *Proceedings of TASE 2012: 6th International Conference on Theoretical Aspects of Software Engineering*. Volume 0, pp. 167–174. IEEE Computer Society, 2012.