DPF Editor: A Multi-Layer Diagrammatic (Meta)Modelling Environment

Yngve Lamo, Xiaoliang Wang, Florian Mantz
Bergen University College, Norway
[yla,xwa,fma]@hib.no

Øyvind Bech
Bergen University College, Norway
oyvind.bech@stud.hib.no

Adrian Rutle
St. Francis Xavier University, Canada
arutle@stfx.ca

Abstract

This paper presents DPF Editor, a diagrammatic (meta)modelling tool for the specification of (meta)models. The tool is a graphical editor for the Diagram Predicate Framework (DPF), that provides a graph based formalisation of (meta)modelling and model transformation. The tool offers fully diagrammatic specification of domain-specific modelling languages. Moreover, DPF Editor supports development of metamodelling hierarchies with an arbitrary number of metalevels; that is, each model at a metalevel can be used as a metamodel for the metalevel below. DPF Editor facilitates the generation of domain-specific diagrammatic editors out of these metamodels. Furthermore, the conformance relations between adjacent metalevels are checked using typing morphisms and validation of diagrammatic constraints. The features of the DPF Editor are illustrated by a running example presenting a metamodelling hierarchy for business process modelling.

1 Introduction

Model-driven engineering (MDE) promotes the use of models as the primary artefacts in the software development process. These models are used to specify, simulate, generate code and maintain the resulting applications. Models can be specified by general-purpose modelling languages such as the Unified Modeling Language (UML) [17], but to fully unfold the potentials of MDE, models are specified by Domain-Specific Languages (DSLs) that are tailored to a specific domain of concern. DSLs are modelling languages where the language primitives consist of domain concepts. Traditionally these domain concepts are specified by a graph based metamodel while the constraints are specified by a text based language such as the Object Constraint Language (OCL) [16]. This mixture of text based and graph based languages is an obstacle for employing MDE especially regarding model transformation [21] and synchronisation of graphical models with their textual constraints [19]. A more practical solution to this problem is a fully graph based approach to the definition of DSLs; i.e. diagrammatic specification of both the metamodel and the constraints.

The availability of tools that facilitate the design and implementation of DSLs is another important factor for the acceptance and adoption of MDE. Moreover, DSLs are required to be intuitive enough for domain experts, while they are formal enough to enable sound model transformation. Since DSLs are defined as metamodels, these tools need to support automatic generation of diagrammatic editors out of these metamodels.

An industrial standard language to describe DSLs is the Meta-Object Facility (MOF) [15] provided by the Object Management Group (OMG). A reference implementation inspired by the MOF standard is Ecore, which is the core language of the Eclipse Modeling Framework (EMF) [22]. This framework uses a two-level metamodelling approach where a model created by the Ecore editor can be used to generate a DSL with a corresponding editor (see Fig. 1a). This editor, in turn, can be used to create instances,

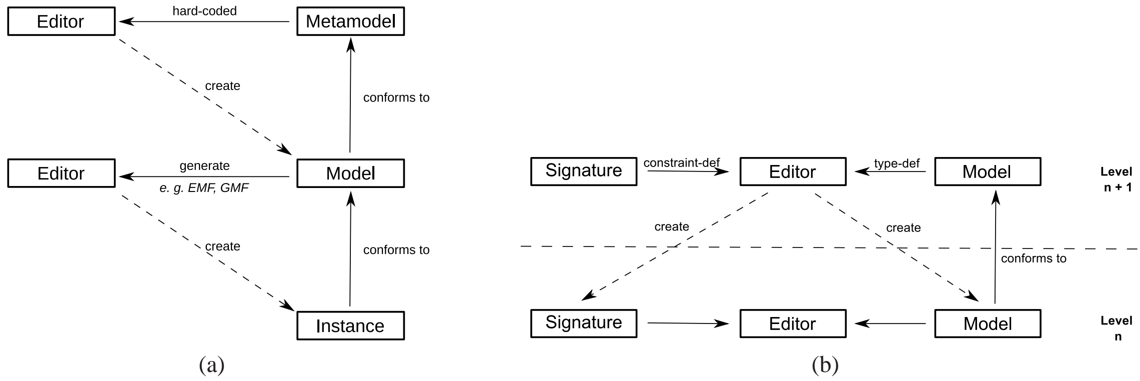


Figure 1: A simplified view of (a) the EMF metamodelling hierarchy, and (b) a generic metamodelling hierarchy as implemented in DPF Editor

however, these instances of the DSL cannot be used to generate other DSLs. That is, the metamodelling process is limited to only two metamodelling levels.

The two-level metamodelling approach has several limitations (see [11, 1] for a comprehensive argumentation). The lack of multi-layer metamodelling support forces DSL designers to introduce type-instance relations in the metamodel. This leads to a mixture of domain concepts with language concepts in the same modelling level. The approach in this paper tackles this issue by introducing a multi-layer metamodelling tool.

This paper presents DPF Editor, a prototype diagrammatic (meta)modelling tool for the specification of (meta)models and the generation of diagrammatic editors from these metamodels. DPF Editor is an implementation of the techniques and methodologies developed in the Diagram Predicate Framework (DPF) [5], that provides a formalisation of (meta)modelling and model transformation based on category theory and graph transformation. DPF Editor supports development of metamodelling hierarchies with an arbitrary number of metalevels; that is, each model at a metalevel can be used as a metamodel for the metalevel below. Moreover, DPF Editor checks the conformance of models to their metamodels by validating both typing and diagrammatic constraints.

The remainder of the paper is organised as follows. Section 2 introduces some basic concepts from DPF. Section 3 gives a brief overview of the tool architecture. Section 4 demonstrates the functionality of the tool in a metamodelling scenario. Section 5 compares DPF Editor with related tools, and finally Section 6 outlines future research and implementation work and concludes the paper.

2 Diagram Predicate Framework

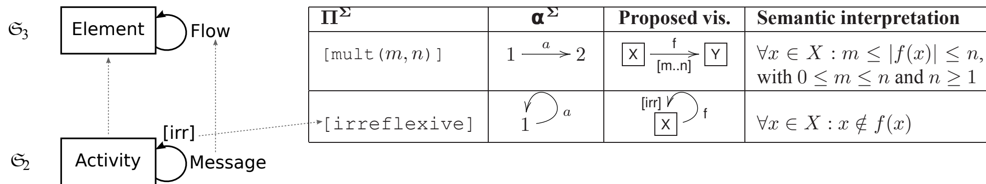
In DPF, models are represented by *(diagrammatic) specifications*. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ [20, 19]. The graph represents the structure of the specification and the atomic constraints represent the restrictions attached to this structure. Atomic constraints are specified by *predicates* from a predefined *(diagrammatic predicate) signature* Σ. A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicates, each having a symbol, an arity (or shape graph), a visualisation and a semantic interpretation (see Table 1).

In the DPF Editor, a DSL corresponds to a diagrammatic editor, which in turn consists of a signature and a metamodel. Each editor can further be used to specify new signatures and metamodels, and thus define new DSLs (see Fig. 1b).

Table 1: The signature Σ used in the metamodelling example

Π^{Σ}	$\alpha^{\Sigma}(\pi)$	**Proposed vis.**	**Semantic Interpretation**
`[mult(`m,n`)]`	$1 \xrightarrow{f} 2$	X —f [m..n]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[irreflexive]`	1 with loop f	X with loop f [irr]	$\forall x \in X : x \notin f(x)$
`[injective]`	$1 \xrightarrow{f} 2$	X —f [inj]→ Y	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[nand]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [nand]	$\forall x \in X :$ $f(x) = \emptyset \vee g(x) = \emptyset$
`[surjective]`	$1 \xrightarrow{f} 2$	X —f [surj]→ Y	$f(X) = Y$
`[jointly-surjective_2]`	$1 \xrightarrow{f} 2$, $3 \xrightarrow{g} 2$	X —f→ Y, Z —g→ Y, [js]	$f(X) \cup g(Z) = Y$
`[xor]`	$1 \xrightarrow{a} 2$, $1 \xrightarrow{b} 3$	X —f→ Y, X —g→ Z, [xor]	$\forall x \in X :$ $(f(x) = \emptyset \vee g(x) = \emptyset)$ and $(f(x) \neq \emptyset \vee g(x) \neq \emptyset)$

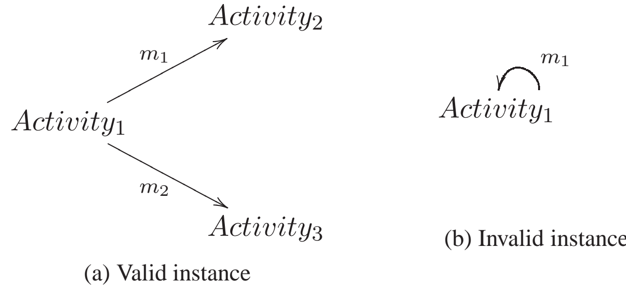
For instance, Fig. 2 shows a specification $\mathfrak{S}_2$ that is compliant with the requirement "activities cannot send messages to themselves". In $\mathfrak{S}_2$, this requirement is forced by the atomic constraint $($`[irreflexive]`$, \delta)$ on the arrow Message. Note that δ is a graph homeomorphism $\delta : \alpha^{\Sigma}($`[irreflexive]`$) \to \mathfrak{S}_2$ specifying which part of $\mathfrak{S}_2$ needs to satisfy the `[irreflexive]` predicate.



Π^{Σ}	α^{Σ}	Proposed vis.	Semantic interpretation
`[mult(`m,n`)]`	$1 \xrightarrow{a} 2$	X —f [m..n]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[irreflexive]`	1 with loop a	X with loop f [irr]	$\forall x \in X : x \notin f(x)$

Figure 2: The specifications $\mathfrak{S}_2$, $\mathfrak{S}_3$ and the signature Σ

The semantics of nodes and arrows of the underlying graph of a specification has to be chosen in a way that is appropriate for the corresponding modelling environment [20, 19]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $X \xrightarrow{f} Y$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of Y is the set of all subsets of Y; i.e. $\wp(Y) = \{A \mid A \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by $(f;g)(x) := \bigcup \{g(y) \mid y \in f(x)\}$.



(a) Valid instance

(b) Invalid instance

Figure 3: (a) A valid instance of $\mathfrak{S}_2$. (b) An invalid instance of $\mathfrak{S}_2$ that violates the atomic constraint ([irreflexive]

The semantics of a specification is defined by the set of its instances (I,ι) [8]. An instance (I,ι) of $\mathfrak{S}$ is a graph I together with a graph homomorphism $\iota : I \to S$ that satisfies the atomic constraints $C^{\mathfrak{S}}$. To check that an atomic constraint is satisfied in a given instance of $\mathfrak{S}$, it is enough to inspect the part of $\mathfrak{S}$ that is affected by the atomic constraint [19]. In this way, an instance of the specification is inspected first to check that the typing is correct, then to check that every constraint in the specification is satisfied.

In DPF, two kinds of conformance relations are distinguished: *typed by* and *conforms to*. A specification $\mathfrak{S}_i$ at metalevel i is said to be typed by a specification $\mathfrak{S}_{i+1}$ at metalevel $i+1$ if there exists a graph homomorphism $\iota_i : S_i \to S_{i+1}$, called the typing morphism, between the underlying graphs of the specifications. A specification $\mathfrak{S}_i$ at metalevel i is said to conform to a specification $\mathfrak{S}_{i+1}$ at metalevel $i+1$ if there exists a typing morphism $\iota_i : S_i \to S_{i+1}$ such that (S_i,ι_i) is a valid instance of $\mathfrak{S}_{i+1}$; i.e. such that ι_i satisfies the atomic constraints $C^{\mathfrak{S}_{i+1}}$.

For instance, Fig. 2 shows a specification $\mathfrak{S}_2$ that conforms to a specification $\mathfrak{S}_3$. That is, there exists a typing morphism $\iota_2 : S_2 \to S_3$ such that (S_2,ι_2) is a valid instance of $\mathfrak{S}_3$. Note that since $\mathfrak{S}_3$ does not contain any atomic constraints, the underlying graph of $\mathfrak{S}_2$ is a valid instance of $\mathfrak{S}_3$ as long as there exists a typing morphism $\iota_2 : S_2 \to S_3$. However, Fig. 3 shows two graphs, both typed by the specification $\mathfrak{S}_2$, but only Fig. 3a is a valid instance of $\mathfrak{S}_2$, since the graph in Fig. 3b violates the $(\texttt{[irreflexive]},\delta)$ constraint on the arrow Message.

3 Tool Architecture

DPF Editor has been developed in Java as a plug-in for Eclipse [9]. Eclipse follows a cross-platform architecture that is well suited for tool integration since it implements the Open Services Gateway initiative framework (OSGi). Moreover, it has an ecosystem around the basic tool platform that offers a rich set of plug-ins and APIs that are helpful when implementing modelling tools. In addition, Eclipse technology is widely used in practise and is also employed in commercial products such as the Rational Software Architect (RSA) [12] as well as in open-source products such as the modelling tool TOPCASED [23]. For this reason DPF Editor can easily be integrated into and used together with such tools.

Figure 4 illustrates that DPF Editor basically consists of three components (Eclipse plugins). The first component (Core Model Management Component) provides the core features of the tool: these are the facilities to create, store and validate DPF specifications. This part uses EMF for data storage. This means the DPF Editor contains an internal metamodel that is an Ecore model. As a consequence, each DPF specification is also an instance of this internal metamodel. EMF has been chosen for data storage since it is a de facto standard in the modelling field and guarantees high interoperability with various other tools and frameworks. Therefore, DPF models can be used with e.g. code generation frameworks such as those offered by the Eclipse Model To Text (M2T) project.

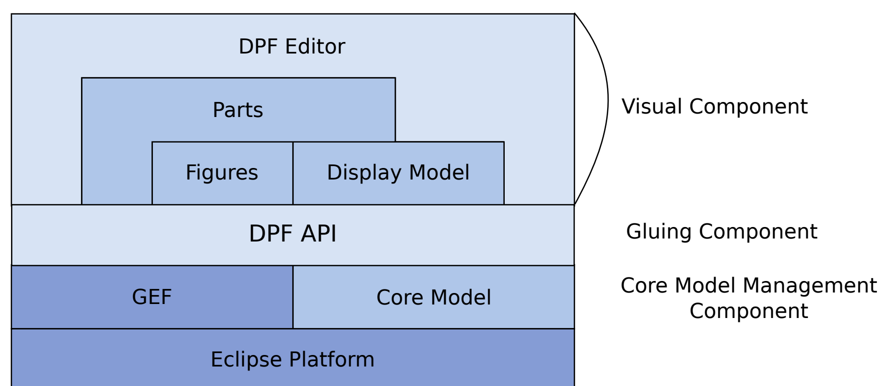


Figure 4: The main component architecture of the DPF Editor plug-in packages

The second component (Visual Component) provides the visual editor of the tool. This component is implemented using the Graphical Editor Framework (GEF). GEF provides technology to create rich graphical editors and views for the Eclipse Workbench. The component mainly consists of classes following GEF's Model-View-Controller (MVC) architecture. There are Figures classes (constituting the view), Display Model classes and controller classes (named Parts in accordance with GEF terminology). Special arrow-routing and display functions have been developed for showing DPF's special kinds of predicates. The third component (Gluing Component) is used as mediator between the first two components. It ties together the functionality and manages file loading, object instantiation and general communication with the Eclipse platform.

4 A Metamodelling Example

This section illustrates the steps of specifying a metamodelling hierarchy using DPF Editor. The example demonstrates specification of a metamodelling hierarchy for business process modelling. First we show the specification of a metamodel using the DPF Editor. We also show the generation of DSL editors by loading an existing metamodel to the tool. Furthermore we present how typing and constraint validation are performed by the tool.

DPF Editor runs inside Eclipse, and to get started, we activate the editor by selecting a project folder and invoking an Eclipse-type wizard for creating a new DPF Specification Diagram. The tool will be pre-loaded with a set of predicates corresponding to the signature shown in Table 1.

We start the metamodelling process by configuring the tool with the DPF Editor's default metamodel $\mathfrak{S}_4$ consisting of Node and Arrow, that serves as a starting point for metamodelling in DPF Editor. This default metamodel is used as the type graph for the metamodel $\mathfrak{S}_3$ at the highest level of abstraction of the business process metamodelling hierarchy. In $\mathfrak{S}_3$, we introduce the domain concepts Elements and Control, that are typed by Node (see Fig. 5). We also introduce Flow, NextControl, ControlIn and ControlOut, that are typed by Arrow. The typing of this metamodel to the default metamodel is guaranteed by the fact that the tool allows only creation of specifications in which each specification element is typed by Node or Arrow. One requirement for process modelling is that "each control should have at least one incoming arrow from an element or another control"; this is specified by adding the `[jointly-surjective_2]` constraint on the arrows ControlIn and NextControl. Another requirement is that "each control should be followed by either another control or by an element, not both"; this is specified by the `[XOR]` constraint on the arrows ControlOut and NextControl. We save this specification in a file called `process_m3.dpf`, with "m3" reflecting the level to which it belongs.

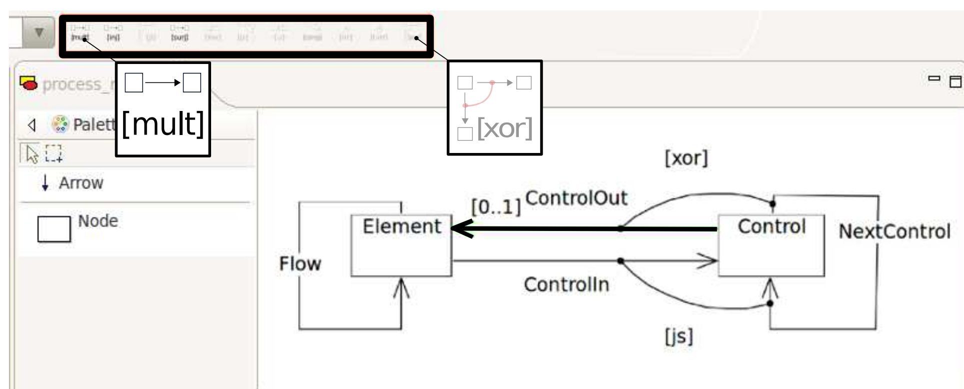


Figure 5: DPF Editor configured with the default metamodel consisting of Node and Arrow, and the signature Σ from Table 1 indicated with a blod black rectangle; showing also the specification $\mathfrak{S}_3$ under construction; note that the bold black arrow ControlOut is selected, therefore the predicates that have arrow as their arity are enabled in the signature bar.

Now, we will illustrate how we can generate an editor from the existing specification $\mathfrak{S}_3$. This is achieved by invoking the wizard for creating a new DPF Specification Diagram once more. This time, in addition to specify that our file shall be called `process_m2.dpf`, we also specify that the file `process_m3.dpf` shall be used as the metamodel for our new specification $\mathfrak{S}_2$. We still use the same signature from Table 1 with this new editor. Note that the tool palette in Fig. 6 contains buttons for each specification element defined in Fig. 5. In `process_m2.dpf` we will define a specification $\mathfrak{S}_2$ which is compliant with the following requirements:

1. Each *activity* may send *message*s to one or more *activities*
2. Each *activity* may be *sequenced* to another *activity*
3. Each *activity* may be connected to at most one *choice*
4. Each *choice* must be connected to at least two *conditions*
5. Each *activity* may be connected either to a *choice* or to another *activity*, but not both.
6. Each *choice* must have exactly one *activity* connected to it
7. Each *condition* must be connected to exactly one *activity*
8. Each *activity* must have exactly one *condition* connected to it
9. An *activity* cannot send messages to itself
10. An *activity* cannot be sequenced to itself

We will explain now how some of the requirements above are specified in $\mathfrak{S}_2$. The requirements 1 and 2 are specified by introducing Activity that is typed by Element, as well as Message and Sequence that are typed by Flow. The requirement 5 is specified by adding the constraint `[nand]` on the arrows Sequence and Choice. The requirement 6 is specified by adding the constraints `[injective]` and `[surjective]` on ChoiceIn. The requirements 9 and 10 are specified by adding the constraint `[irreflexive]` on Message and Sequence, respectively.

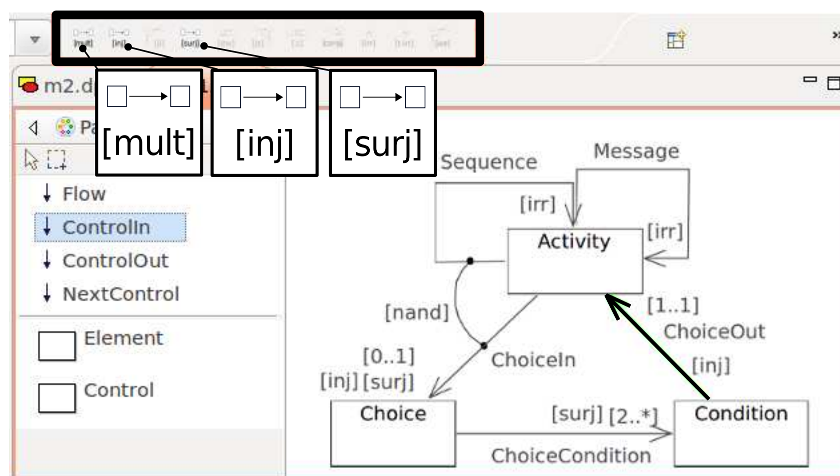


Figure 6: DPF Editor configured with the specification $\mathfrak{S}_3$ from Fig. 5 as metamodel, and the signature Σ from Table 1 indicated with a bold black rectangle; showing also the specification $\mathfrak{S}_2$ under construction

The conformance relation between $\mathfrak{S}_2$ and $\mathfrak{S}_3$ is checked in two steps. Firstly, $\mathfrak{S}_2$ specification is correctly typed over its metamodel by construction. The DPF Editor actually checks that there exists a graph homomorphism from the specification to its metamodel while creating a specification. For instance, when we create the ChoiceIn arrow of type ControlIn, the tool ensures that the source and target of ChoiceIn are typed by Element and Control, respectively. Secondly, the constraints are checked by corresponding validators during creation of specifications. In Fig. 6 we see that all constraints specified in $\mathfrak{S}_3$ are satisfied by $\mathfrak{S}_2$. However, Fig. 7 shows a specification which violates some of the constraints of $\mathfrak{S}_3$, e.g. the `[XOR]` constraint on the arrows ControlOut and NextControl in $\mathfrak{S}_3$ is violated by the arrow WrongArrow in $\mathfrak{S}_2$. The constraint is violated since Condition – that is typed by Control – is followed by both a Choice, and an Activity, violating the requirement "each control should be followed by either another control or by an element, not both". This violation will be indicated in the tool by a message (or a tip) in the status bar.

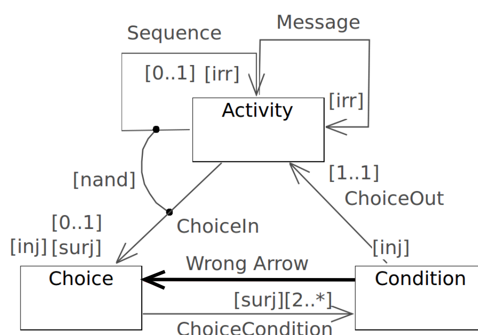


Figure 7: A specification violating `[XOR]` constraint on the arrows ControlOut and NextControl in $\mathfrak{S}_3$

We can now repeat the previous step and load the editor with the specification $\mathfrak{S}_2$ as metamodel, by choosing `process_m2.dpf` as a metamodel. This editor is then used to specify other specifications located at the metalevel M_1. One such specification $\mathfrak{S}_1$ is shown in Fig. 8. Note that this time the tool palette in Fig. 8 contains buttons for each specification element defined in Fig. 6. For this tool palette (not

shown in Fig. 8) we have chosen a concrete syntax for process modelling with special visual effects for model elements. For instance model elements typed by Sequence and Message are visualised as arrows and dashed arrows, respectively, and, model elements typed by Condition and Choice are visualised as diamonds and circles, respectively.

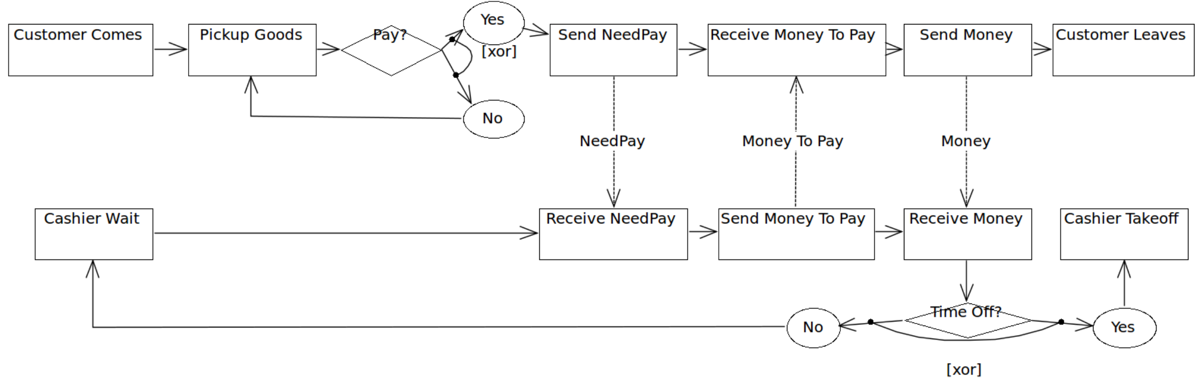


Figure 8: A sample business process model $\mathfrak{S}_1$ for online purchasing, specified by the DPF Editor configured with the specification $\mathfrak{S}_2$ as metamodel

Finally, we may use predicates from the signature to add constraints to $\mathfrak{S}_1$, and, we may use it as a metamodel for another modelling level. This pattern could be repeated as deep as it is necessary for the metamodelling hierarchy, however, in this particular running example we stop at this level, and will eventually generate code from $\mathfrak{S}_1$. In that case, the constrains added to this specification may be used to generate invariants in the code.

5 Related Work

There is an abundance of visual modelling tools available, both as open-source software and as closed-source commercial products. Some of these tools also possess metamodelling features, letting the user specify a metamodel and then use this metamodel to create a new editor. We will now give a short description of some of the most popular metamodelling tools and shortly discuss how they treat multi-level metamodelling. We will also mention how constraints are represented in the metamodelling process.

The Visual Modeling and Transformation System (VMTS) is an n-layer metamodelling environment that supports editing models according to their metamodels [14]. VMTS allows for an arbitrary number of (meta)modelling layers, but has no support for a completely graph based constraint specification language, as it uses OCL for the specification of constraints. It runs on the Microsoft .NET framework.

AToM^3 (A Tool for Multi-formalism and Meta-Modeling) is a tool for multi-paradigm modelling [2, 7]. The two main tasks of AToM^3 are metamodelling and model-transformation. Formalisms and models are described as graphs. From the metamodel of a formalism, AToM^3 can generate a tool that lets the user create and edit models described in the specified formalism. Some of the metamodels currently available are: Entity-Relationship, GPSS, Deterministic and Non-Deterministic Finite State Automata, Data Flow Diagrams etc. AToM^3 is freely available. The tool does not allow for an arbitrary number of (meta)modelling layers, nor is there support for a completely graph based constraint specification language. The tool is implemented in Python and runs on most platforms.

The Generic Modeling Environment (GME) [13] is a configurable toolkit for creating domain-specific modelling and program synthesis environments. The configuration is accomplished through

Table 2: Comparison of DPF Editor to other metamodelling tools, EVL stands for Epsilon Validation Language, and the current predefined validator in DPF is implemented in Java

Tool	Layers	Diag. Const.	Const. Lang.	Platform	Visual UI
EMF/GMF	2		OCL, EVL, Java	Java VM	✓
VMTS	∞		OCL	Windows	✓
AToM^3	2		OCL, Python	Python, Tk/tcl	✓
GME	2		OCL	Windows	✓
metaDepth	∞		EVL	Java VM	
DPF Editor	∞	✓	Predefined validator	Java VM	✓

metamodels specifying the modelling paradigm (modelling language) of the application domain [10]. The GME metamodelling language is based on the UML class diagram notation and OCL constraints. Metamodels specifying the modelling paradigm are edited in the tool's editor and saved to file. New editors can then be instantiated, based on the newly generated metamodels. In order to simplify the editing process, both models and metamodels are edited in the same environment. Model visualisation is customisable through built-in decorator interfaces. All GME modelling languages provide type inheritance, and GME supports various concepts for modelling, including hierarchy, multiple aspects, sets, references, and explicit constraints. The tool does not allow for an arbitrary number of (meta)modelling layers, nor is there support for a completely graph based constraint specification language. GME's architecture is based on Microsoft Component Object Model (COM), making it extensible by any language that supports COM. The drawback of this approach is that GME only runs on the Microsoft Windows platform.

The metaDepth [6] framework is a framework for multi-level metamodelling. The system permits building systems with an arbitrary number of metalevels through deep metamodelling. The framework allows the specification and evaluation of derived attributes and constraints across multiple metalevels, linguistic extensions of ontological instance models, transactions, and hosting different constraint and action languages. At present, the framework supports only textual specifications; it does not yet support diagrammatic syntax. However, there is some work in progress on integrating DPF with metaDepth that aims to give a graph based formalisation of metaDepth, and deep metamodelling in general.

Table 2 summarises the comparison of some popular metamodelling tools with DPF Editor. Note that DPF Editor is the only tool that supports fully diagrammatic metamodelling. The table also shows that only few tools that support multi-level modelling, especially combined with platform independence.

6 Conclusion and Future Work

In this article, the prototype modelling tool DPF Editor was presented. Note that although DPF is well established on the theoretical level, this is the first time its corresponding prototype tool is published. The tool is developed in Java and runs as a plug-in on the Eclipse platform. DPF Editor supports fully diagrammatic metamodelling as proposed by the DPF Framework. The functionality of the tool has been illustrated by specifying a metamodelling hierarchy for business process modelling. It has been shown how the editor's tool palette can be configured for a given domain by using a specific metamodel. To ensure correct typing of the edited models the tool uses graph homomorphism. Moreover, it implements a validation mechanism that checks instances against all the constraints that are specified by the metamodel. We have also shown how models created in the tool can be used as metamodels at an arbitrary number of metamodelling levels. The authors are not aware of other EMF based tools that facilitate

multi-level metamodelling.

The tool was used as an experimental prototype in a graduate course in MDE at Bergen University College (HiB). The students participated in a field experiment designed for the testing of DPF Editor. This experiment gave positive user feedback from participants external to the development project.

Many directions for further work still remains unexplored, other are currently in the initial development phases. We shall only mention the most prominent here:

Code generation. The real utility for an end user of DPF Editor will become manifest when an actual running system can be generated from specifications. We have already done some introductory work on code generation [4], and further work by other graduate students in this area has already been commenced. Several solutions can be envisaged, for example Java code generation for class modelling or SQL code generation for data modelling.

Signature editor. Today's plug-in offer a single pre-defined signature for modelling use. This signature, as well as the functionality for semantic validation, is hard-coded, and future users of DPF Editor will probably wish to define their own. A two-stage solution for this can be suggested:

- Create a simple signature editor that lets the user compile signatures by choosing between pre-defined predicates
- Expand on this solution by enabling the user to edit individual predicates.

Configurable concrete syntax. As the system exists today, all diagram (nodes, arrows and constraints) visualisations are hardcoded in the editor code. A desirable extension would be to make visualisations more decoupled from the rest of the Display Model than is the current situation. This would involve a configurable and perhaps directly editable *concrete syntax* [3].

Layout and routing. Automated layout seems to become an issue when dealing with medium-sized to large diagrams. There seems to be a big usability gain to be capitalised on in this matter. Today's editor contains a simple routing algorithm, based on GEF's `ShortestPathConnectionRouter` class. The problem of finding routing algorithms that produce easy-readable output is a focus of continuous research [18], and this problem applied to DPF Editor can probably be turned into a separate research task.

In addition to these areas, development to utilise the core functionality of DPF Editor as a base for model transformation and (meta)model evolution is on the horizon, reflecting the theoretical foundations that are being laid down within the DPF research community.

References

[1] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Transactions on Modeling and Computer Simulation*, 12(4):290–321, 2002.

[2] AToM^3: A Tool for Multi-formalism and Meta-Modelling. *Project Web Site*. `http://atom3.cs.mcgill.ca/`.

[3] Thomas Baar. Correctly Defined Concrete Syntax for Visual Modeling Languages. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of MoDELS 2006: 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

[4] Øyvind Bech and Dag Viggo Lokøen. *DPF to SHIP Validator Proof-of-Concept Transformation Engine*. `http://dpf.hib.no/code/transformation/dpf_to_shipvalidator.py`.

[5] Bergen University College and University of Bergen. *Diagram Predicate Framework (DPF) Web Site*. http://dpf.hib.no/.

[6] Juan de Lara and Esther Guerra. Deep Meta-modelling with METADEPTH. In Jan Vitek, editor, *Proceedings of TOOLS 2010: 48th International Conference on Objects, Components, Models and Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2010.

[7] Juan de Lara and Hans Vangheluwe. Using AToM^3 as a Meta-CASE Tool. In *Proceedings of ICEIS 2002: 4th International Conference on Enterprise Information Systems*, pages 642–649, Ciudad Real, Spain, April 2002.

[8] Zinovy Diskin and Uwe Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *Proceedings of ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*, volume 203/6 of *Electronic Notes in Theoretical Computer Science*, pages 19–41, Amsterdam, The Netherlands, 2008. Elsevier.

[9] Eclipse Platform. *Project Web Site*. http://www.eclipse.org.

[10] GME: Generic Modeling Environment. *Project Web Site*. http://www.isis.vanderbilt.edu/Projects/gme/.

[11] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008.

[12] IBM. *Rational Software Architect*. http://www-01.ibm.com/software/awdtools/architect/swarchitect/.

[13] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *Proceedings of WISP 2001: Workshop on Intelligent Signal Processing*, volume 17, pages 82–83. ACM Press, 2001.

[14] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Constraint Validation Support in Visual Model Transformation Systems. *Acta Cybernetica*, 17(2):339–357, 2005.

[15] Object Management Group. *Meta-Object Facility Specification*, January 2006. http://www.omg.org/spec/MOF/2.0/.

[16] Object Management Group. *Object Constraint Language Specification*, February 2010. http://www.omg.org/spec/OCL/2.2/.

[17] Object Management Group. *Unified Modeling Language Specification*, May 2010. http://www.omg.org/spec/UML/2.3/.

[18] Tobias Reinhard, Christian Seybold, Silvio Meier, Martin Glinz, and Nancy Merlo-Schett. Human-Friendly Line Routing for Hierarchical Diagrams. In *Proceedings of ASE 2006: 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 273–276, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[20] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In Manuel Oriol and Bertrand Meyer, editors, *Proceedings of TOOLS 2009: 47th International Conference on Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 37–56. Springer, 2009.

[21] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Formalisation of Constraint-Aware Model Transformations. In David Rosenblum and Gabriele Taentzer, editors, *Proceedings of FASE 2010: 13th International Conference on Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2010.

[22] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008.

[23] TOPCASED. *Project Web Site*. http://www.topcased.org.