DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment

Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle

Abstract. This paper presents the DPF Workbench, a diagrammatic tool for domain specific modelling. The tool is an implementation of the basic ideas from the Diagram Predicate Framework (DPF), which provides a graph based formalisation of (meta)modelling and model transformations. The DPF Workbench consists of a specification editor and a signature editor and offers fully diagrammatic specification of domain-specific modelling languages. The specification editor supports development of metamodelling hierarchies with an arbitrary number of metalevels; that is, each model can be used as a metamodel for the level below. The workbench also facilitates the automatic generation of domain-specific specification editors out of these metamodels. Furthermore, the conformance relations between adjacent metalevels are dynamically checked by the use of typing morphisms and constraint validators. The signature editor is a new component that extends the DPF Workbench with functionality for dynamic definition of predicates. The syntax of the predicates are defined by a shape graph and a graphical icon, and their semantics are defined by validators. Those predicates are used to add constrains on the underlying graph. The features of the DPF Workbench are illustrated by a running example presenting a metamodelling hierarchy for workflow modelling in the health care domain.

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

Yngve Lamo · Xiaoliang Wang · Florian Mantz
Bergen University College, Norway
e-mail: {yla,xwa,fma}@hib.no

Wendy MacCaull · Adrian Rutle
St. Francis Xavier University, Canada
e-mail: {wmaccaul,arutle}@stfx.ca

R. Lee (Ed.): Computer and Information Science 2012, SCI 429, pp. 37–52.
springerlink.com

by general-purpose modelling languages such as the Unified Modeling Language (UML) [20], but to fully unfold the potentials of MDE, models are specified by Domain-Specific Modelling Languages (DSMLs), each tailored to a specific domain of concern [11]. DSMLs are modelling languages where the language primitives consist of domain concepts. Traditionally such 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) [19]. This mixture of text based and graph based languages is an obstacle for employing MDE especially with regard to model transformations [25] and synchronisation of graphical models with their textual constraints [23]. A more practical solution to this problem is a fully graph based approach to the definition of DSMLs; i.e., diagrammatic specification of both the metamodel and the constraints [24].

The availability of tools that facilitate the design and implementation of DSMLs is an important factor for the acceptance and adoption of MDE. DSMLs are required to be intuitive enough for domain experts whereas they have a solid formal foundation which enables automatic verification and sound model transformations. Since DSMLs are defined by metamodels, these tools need to support automatic generation of specification editors out of metamodels.

An industrial standard language to describe DSMLs is the Meta-Object Facility (MOF) [18] 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 Modelling Framework (EMF) [26]. This framework uses a two-level metamodelling approach where a model created by the Ecore editor can be used to generate a DSML with a corresponding editor (see Fig. 1a). This editor, in turn, can be used to create instances; however, the instances of the DSML cannot be used to generate other DSMLs. That is, the metamodelling process is limited to only two user-defined metamodelling levels.

The two-level metamodelling approach has several limitations (see [13, 1] for a comprehensive argumentation). The lack of multi-layer metamodelling support

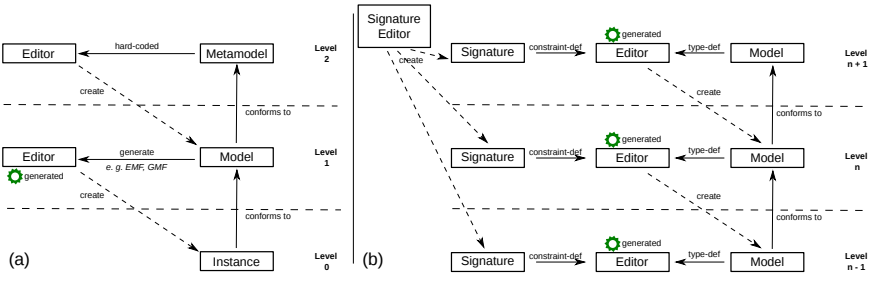


Fig. 1 A simplified view of (a) the EMF metamodelling hierarchy, and (b) a generic metamodelling hierarchy as implemented in the DPF Workbench

forces DSML 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 the DPF Workbench, a prototype diagrammatic (meta)modelling tool for the specification of diagrammatic signatures, (meta)models, and the generation of specification editors (see Fig. 1b). The DPF Workbench 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 transformations based on category theory and graph transformations. The DPF Workbench supports the development of metamodelling hierarchies by providing an arbitrary number of metalevels; that is, each model at a metalevel can be used as a metamodel for the metalevel below. Moreover, the DPF Workbench checks the conformance of models to their metamodels by validating both typing morphisms and diagrammatic constraints. DPF Workbench extends the DPF Editor [15] with a signature editor which is used to define new domain specific predicates (syntax) and their corresponding validators (semantics).

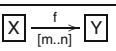
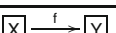
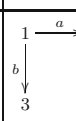
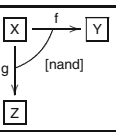
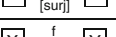
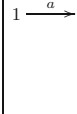
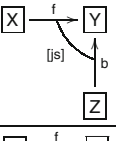
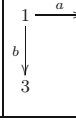
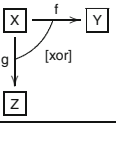
The functionality of the DPF Workbench is demonstrated by specifying a metamodelling hierarchy for health workflows. Health services delivery processes are complicated and are frequently developed from regional or national guidelines which are written in natural language. Having good models of these processes is particularly valuable for several reasons (1) the modelling process which must be done in conjunction with (health) domain experts clarifies the meaning of the guidelines and has, in a number of situations, found ambiguities or inconsistencies in the guidelines; (2) graphical display of a process makes it easy for the (clinicians) domain experts to understand; (3) formal descriptions can be analysed for their behavioural characteristics via model checking; and, (4) the models can drive an executable workflow engine to guide the actual process in health care settings. The use of MDE technology is especially valuable because guidelines may be updated or changed every few years so the model and associated workflow must be redeveloped; moreover, though compliance with guidelines is required across a province or country, individual health districts, indeed individual hospitals or other service settings (clinics, etc.) will have processes that are specific to their setting so the overall process must be customised for the local setting. With MDE once the model is written and analysed for correctness the executable code is generated automatically. Moreover the abstraction required for development of abstract models makes it easier to involve domain experts in the development process. MDE transformation techniques can be used to generate code suitable for model checkers to verify behavioural characteristics of a workflow model, an important feature for a safety critical applications such as health care. Health care costs are rising dramatically worldwide; better outcomes for the patient as well as enhanced efficiencies have been shown to result from better process (workflow) definitions.

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 Workbench 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* $C^{\mathfrak{S}}$ [24, 23]. 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) 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).

Table 1 The signature Σ used in the metamodelling example

Π^{Σ}	$\alpha^{\Sigma}(\pi)$	Visualisation	Semantics
`[mult(m,n)]`			$\forall x \in X : m \leq \lvert f(x)\rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[irreflexive]`			$\forall x \in X : x \notin f(x)$
`[injective]`			$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[nand]`			$\forall x \in X :$ $f(x) = \emptyset \vee g(x) = \emptyset$
`[surjective]`			$f(X) = Y$
`[jointly-surjective_2]`			$f(X) \cup g(Z) = Y$
`[xor]`			$\forall x \in X :$ $(f(x) = \emptyset \vee g(x) = \emptyset)$ and $(f(x) \neq \emptyset \vee g(x) \neq \emptyset)$

In the DPF Workbench, a DSML corresponds to a specification editor, which in turn consists of a signature and a metamodel. A specification editor can be used to specify new metamodels, and thus define new DSMLs (see Fig. 1b).

Next we show a specification Fig. 2 is an example of a specification $\mathfrak{S}_2$ that ensures that "activities cannot send messages to themselves". In $\mathfrak{S}_2$, this requirement is forced by the atomic constraint $(\texttt{[irreflexive]}, \delta)$ on the arrow Message. Note that δ is a graph homomorphism $\delta : (\ 1 \overset{a}{\circlearrowright}\ ) \to (\ \text{Activity} \overset{\text{Message}}{\circlearrowright}\ )$ specifying the part of $\mathfrak{S}_2$ to which the `[irreflexive]` predicate is added.

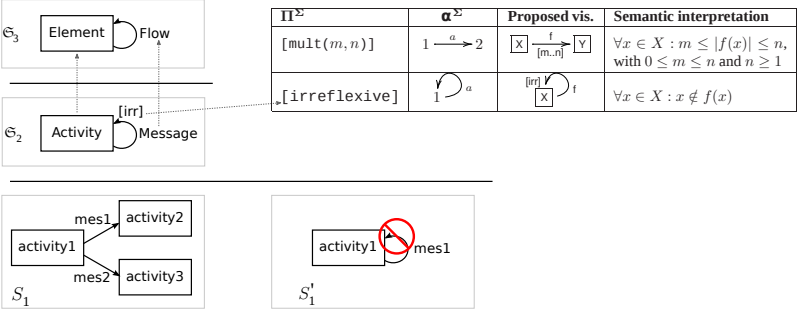


Fig. 2 The specifications $\mathfrak{S}_2, \mathfrak{S}_3$, the signature Σ, a valid instance S_1 of $\mathfrak{S}_2$, and an invalid instance S_1' of $\mathfrak{S}_2$ that violates the irreflexivity constraint

The semantics of the underlying graph of a specification must be chosen in a way that is appropriate for the corresponding modelling environment [24, 23]. 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 $(f;g)(x) := \bigcup\{g(y) \mid y \in f(x)\}$.

The semantics of a specification is defined by the set of its instances (I, ι) [9]. 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 S that is affected by the atomic constraint [23]. 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. For example, Fig. 2 shows two graphs S_0, S_0', both typed by the specification $\mathfrak{S}_2$, but only S_0 is a valid instance of $\mathfrak{S}_2$, since S_0' violates the $(\texttt{[irreflexive]}, \delta)$ constraint on Message by having a circular (reflexive) arrow of type Message.

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 $\iota_2 : S_2 \to S_3$.

3 Tool Architecture

The DPF Workbench has been developed in Java as a plug-in for Eclipse [10]. 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 technologies are widely used in practise and are also employed in commercial products such as the Rational Software Architect (RSA) [14] as well as in open-source products such as the modelling tool TOPCASED [27]. For this reason the DPF Workbench can be integrated into such tools easily and used together with them.

Figure 3 illustrates that the DPF Workbench basically consists of three components (Eclipse plugins). The bottom component (the 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 Workbench 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. Recently an adapter for Xpand (M2T) has been added to the workbench offering native support for DPF specifications. This means Xpand templates can use the DPF metamodelling hierarchy in addition to the one which is given by EMF.

The top component (the Visual Component) provides the visual editors, i.e., specification editors and signature editors. This component is implemented using the Graphical Editing 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 the GEF terminology). Special arrow-routing and display functions have been developed for showing DPF's special kinds of predicates.

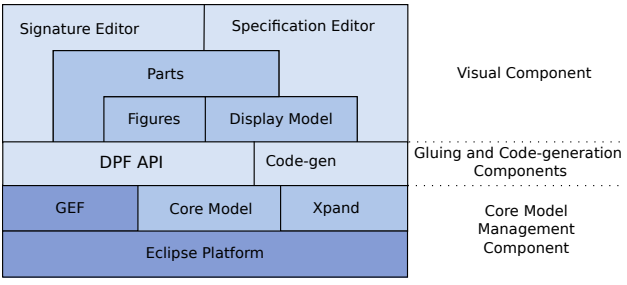


Fig. 3 The main component architecture of the DPF Workbench plug-in packages

The middle component (the 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 the DPF Workbench. The example demonstrates a metamodelling hierarchy that is used to specify a workflow for the treatment of Cancer Related Pain [6]. First we use the signature editor to define the signature that is used in the example. Then we show how to specify a metamodel using the DPF Workbench. We also show the generation of specification editors for DSML by loading an existing metamodel and an existing signature into the tool. Furthermore we present how type checking and constraint validation are performed by the workbench.

4.1 Creating Signatures

We first explain how we create a new project in the DPF Workbench and define the signature; i.e., the predicates that will be available in the modelling process. The DPF Workbench runs inside Eclipse, and has its own custom perspective. To get started, we activate the specification editor by selecting a project folder and invoking an Eclipse-type wizard for creating a new DPF Signature. The signature for the metamodelling hierarchy must include the predicates from Table 1. We use the signature editor to define the arity of the predicates, a graphical icon that illustrates the predicates in the DPF Workbench toolbar and the semantics of the predicates. Figure 4 shows how the arity of the `[xor]` predicate is defined. Figure 5 shows how the semantics of the `[xor]` predicate is defined as a Java validator. An example usage of the `[xor]` predicate explained in Section 4.2. Currently the semantics can only be defined by Java validators, but in future, it will be possible to define the semantics also by use of OCL syntax.

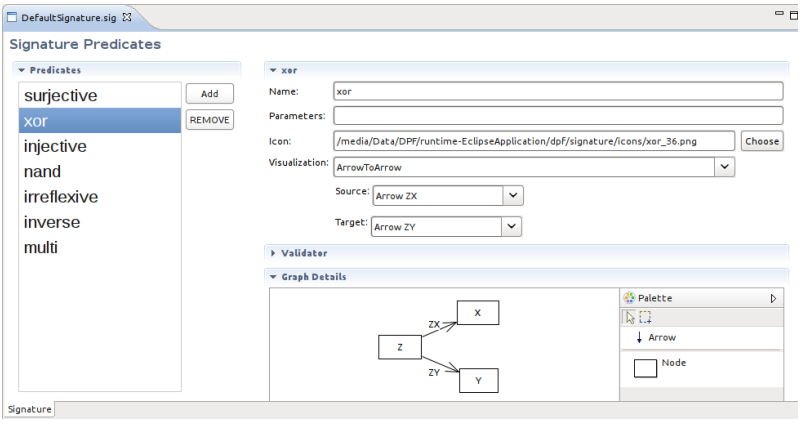


Fig. 4 Definition of the arity and the graphical icon of the `[xor]` predicate

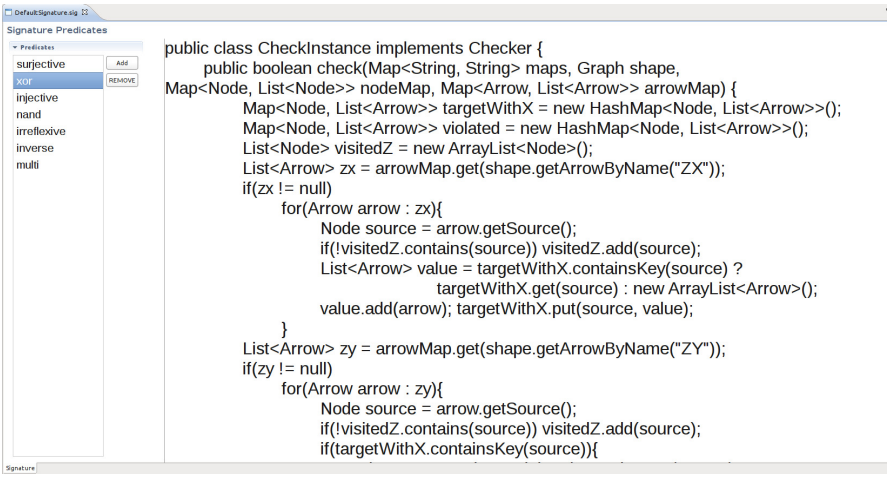


Fig. 5 An excerpt of the definition of the Java validator for the `[xor]` predicate

4.2 Defining Metamodels

After defining all the necessary predicates we load the DPF Workbench with the desired set of predicates corresponding to the signature shown in Table 1.

We start the metamodelling process by configuring the tool with the DPF Workbench's default metamodel $\mathfrak{S}_4$ consisting of Node and Arrow, that serves as a starting point for metamodelling in the DPF Workbench. This default metamodel is used as the type graph for the metamodel $\mathfrak{S}_3$ at the highest level of abstraction of the workflow metamodelling hierarchy. In $\mathfrak{S}_3$, we introduce the domain concepts Elements and Control, that are typed by Node (see Fig. 6). We also introduce Flow,

NextControl, ControlIn and ControlOut, that are typed by Arrow. The typing of this metamodel by 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 by 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 metalevel M_3 to which it belongs.

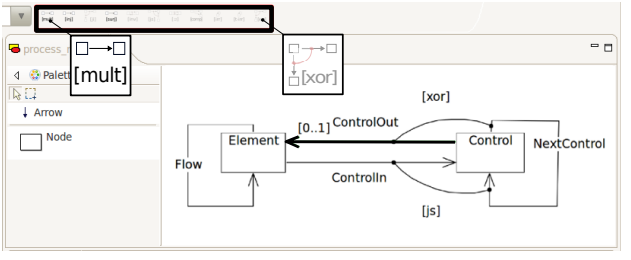


Fig. 6 DPF Workbench configured with the default metamodel consisting of Node and Arrow, and the signature Σ from Table 1 indicated with a bold black rectangle; showing also the specification $\mathfrak{S}_3$ under construction; note that the bold black arrow ControlOut is selected, therefore the predicates that have $1 \to 2$ as their arity are enabled in the signature bar.

4.3 Generating Specification Editors from Metamodels

In this section, we illustrate how a specification editor can be generated from the existing specification $\mathfrak{S}_3$. This is achieved by again invoking the wizard for creating a new DPF Specification Diagram. This time, in addition to specifying 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 use the signature from Table 1 with this new specification editor. Note that the tool palette in Fig. 7 contains buttons for each specification element defined in Fig. 6. 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. Exactly one *activity* must be connected to each *choice*

7. Each *condition* must be connected to exactly one *activity*
8. An *activity* cannot send messages to itself
9. 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 the Activity node that is typed by Element, as well as Message and Sequence arrows 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 8 and 9 are specified by adding the constraint `[irreflexive]` on Message and Sequence, respectively.

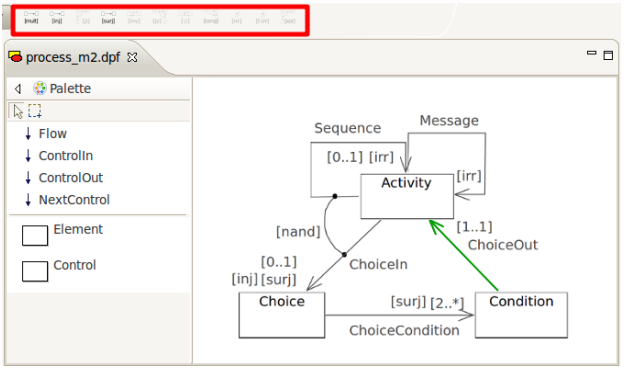


Fig. 7 The DPF Workbench configured with the specification $\mathfrak{S}_3$ from Fig. 6 as metamodel, and the signature Σ from Table 1 indicated with a bold black rectangle; the specification $\mathfrak{S}_2$ under construction is also shown.

4.4 Conformance Checks

The conformance relation between $\mathfrak{S}_2$ and $\mathfrak{S}_3$ is checked in two steps. Firstly, the specification $\mathfrak{S}_2$ is correctly typed over its metamodel by construction. The DPF Workbench actually checks that there exists a graph homomorphism from the specification to its metamodel while creating the 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. 7 we see that all constraints specified in $\mathfrak{S}_3$ are satisfied by $\mathfrak{S}_2$. However, Fig. 8 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.

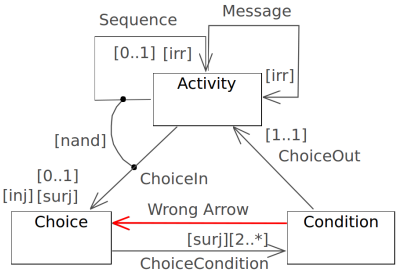


Fig. 8 A specification violating the `[xor]` constraint on the arrows ControlOut and NextControl in $\mathfrak{S}_3$

4.5 Further Modelling Layers

We can now repeat the previous step and load the specification editor with the specification $\mathfrak{S}_2$ (by choosing `process_m2.dpf`) as metamodel. This editor is then used to specify the workflow model at the metalevel M_1. The example is taken from the Guidelines for the Management of Cancer-Related Pain in Adults [6]. This guideline outlines the procedure to manage a patient's pain. The Pain Assessment activity assesses all causes of pain (total pain), determine pain location(s), pain intensity, and other symptoms. Complete history and all previous analgesics (including opioids) and response to each will be documented in this activity. After assessment, if the patient is currently under any opioid medication then the flow will be either forwarded to the Strong Opioid Regimen1 or Strong Opioid Regimen2 activity, depending on the pain level and current opioid dose. Otherwise, the flow goes to the Non-Opioid or Weak Opioid or Strong Opioid Regimen1 activity. While a patient is taking any Strong Opioid medication, his/her pain intensity is assessed regularly and the dose is adjusted accordingly. If any symptoms for opioid toxicity or other side effects are found, they are managed appropriately. We modelled the workflow in the DPF Editor and ensured that the model is conformant to its metamodel. In [21], the authors modelled the guideline using a different workflow modelling language (called CWML). They monitored some interesting properties involving pain reassessment times and verified some behavioural LTL-properties using an automated translator to a model checker. In that work, the authors did not focus on metamodelling or the conformance aspects.

The guideline is represented as the DPF specification $\mathfrak{S}_1$ in Fig. 9. Note that this time the tool palette contains buttons for each specification element defined in Fig. 7. For this tool palette (not shown in Fig. 9) we have chosen a concrete syntax for process modelling with special visual effects for model elements. For instance, model elements typed by Choice and Condition are visualised as diamonds and circles, respectively. In future, to enhance readability, the specification editor will facilitate other visualisation functionalities like zooming, grouping, etc.

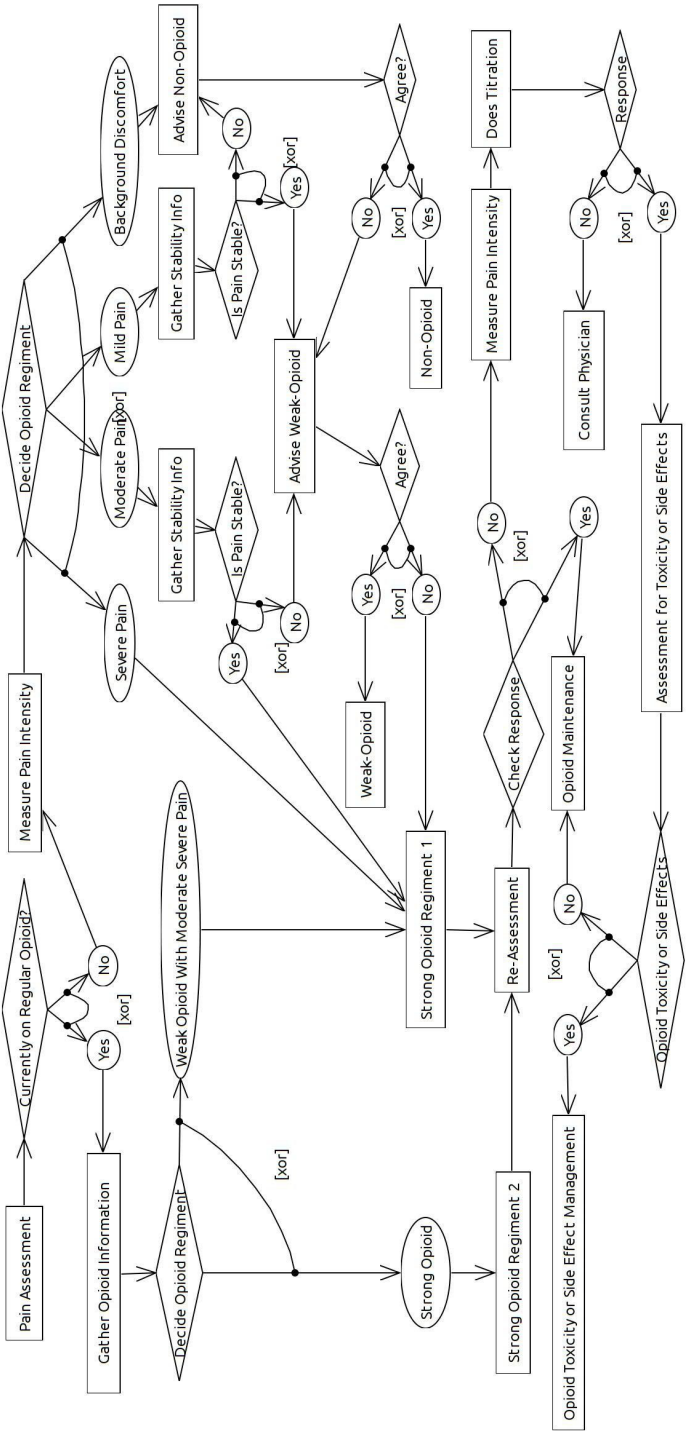


Fig. 9 The Guidelines for the Management of Cancer-Related Pain in Adults

Finally, we may use predicates from the signature to add constraints to $\mathfrak{S}_1$, and, we may use $\mathfrak{S}_1$ as a metamodel for another modelling level. This pattern could be repeated as deep as it is necessary, however, in this example we stop at this level, and will eventually generate the code that is used for model checking of $\mathfrak{S}_1$.

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 users specify a metamodel and then use this metamodel to create a new specification editor. Table 2 summaries the comparison of some popular metamodelling tools with the DPF Workbench.

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

Tool	No. of Layers	Diagrammatic language	Constraint	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	∞	✓	Predefined validator	Java VM	✓

The Visual Modelling and Transformation System (VMTS) supports editing models according to their metamodels [17]. AToM^3 (A Tool for Multi-formalism and Meta-Modelling) is a tool for multi-paradigm modelling [2, 8]; 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, Deterministic and Non-Deterministic Finite State Automata, Data Flow Diagrams, etc.

The Generic Modelling Environment (GME) [16] is a configurable toolkit for creating domain-specific modelling and program synthesis environments. The configuration is accomplished through metamodels specifying the modelling paradigm (modelling language) of the application domain [12]. The GME metamodelling language is based on the UML class diagram notation and OCL constraints.

The metaDepth [7] framework 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; 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.

The table shows that VMTS, metaDepth and DPF Workbench support n-layer metamodelling, while other three tools, AToM^3, GME and EMF/GMF only support two level metamodelling. Most tools use OCL as their constraint language while Java, EVL and Python are alternatives. Those tools have no support for diagrammatic constraints, except the DPF Workbench, which has a dynamic definition of constraint syntax and corresponding semantics by use of Java validators.

6 Conclusion and Future Work

In this paper, we presented the prototype (meta)modelling tool DPF Workbench. The tool is developed in Java and runs as a plug-in on the Eclipse platform. The DPF Workbench supports fully diagrammatic metamodelling as proposed by the DPF Framework. The functionality of the tool has been illustrated by specifying a metamodelling hierarchy for health workflow modelling. Workflow is becoming an increasingly popular paradigm across many domains to articulate process definitions and guide actual processes. While the methods discussed here have been applied to a case study involving workflow for a health care system, they can be used to develop correct and easily customisable executable process definitions for many complex and safety critical systems. It has been shown how the specification editor's tool palette can be configured for a given domain by using a specific metamodel and a specific signature. To ensure correct typing of the edited models the tool uses graph homomorphisms. 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 DPF Workbench also includes a signature editor to dynamically create new predicates and their corresponding semantics.

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 Workbench 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 lately an Xpand adapter is included to the DPF Workbench. The adapter will be used in a further work to automatically generate scripts that will be used for model checking workflows specified by the DPF Workbench.

Configurable concrete syntax. As the system exists today, all diagram (nodes, arrows and constraints) visualisations are hardcoded in the specification 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 specification 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 [22], and this problem applied to DPF Workbench can probably be turned into a separate research task.

In addition to these areas, development to utilise the core functionality of DPF Workbench 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. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation 12(4), 290–321 (2002), doi:10.1145/643120.643123
2. AToM^3: A Tool for Multi-formalism and Meta-Modelling: Project Web Site, `http://atom3.cs.mcgill.ca/`
3. Baar, T.: Correctly Defined Concrete Syntax for Visual Modeling Languages. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 111–125. Springer, Heidelberg (2006)
4. Bech, Ø., Lokøen, D.V.: 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 Web Site, `http://dpf.hib.no/`
6. Broadfield, L., Banerjee, S., Jewers, H., Pollett, A., Simpson, J.: Guidelines for the Management of Cancer-Related Pain in Adults. Supportive Care Cancer Site Team, Cancer Care Nova Scotia (2005)
7. de Lara, J., Guerra, E.: Deep Meta-modelling with METADEPTH. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer, Heidelberg (2010)
8. de Lara, J., Vangheluwe, H.: Using AToM^3 as a Meta-CASE Tool. In: Proceedings of ICEIS 2002: 4th International Conference on Enterprise Information Systems, Ciudad Real, Spain, pp. 642–649 (2002)
9. Diskin, Z., Wolter, U.: A Diagrammatic Logic for Object-Oriented Visual Modeling. In: Proceedings of ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory, vol. 203(6), pp. 19–41. Elsevier (2008), doi:10.1016/j.entcs.2008.10.041
10. Eclipse Platform: Project Web Site, `http://www.eclipse.org`
11. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
12. GME: Generic Modeling Environment: Project Web Site, `http://www.isis.vanderbilt.edu/Projects/gme/`
13. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley (2008)

14. IBM: Rational Software Architect, http://www-01.ibm.com/software/awdtools/architect/swarchitect/
15. Lamo, Y., Wang, X., Mantz, F., Bech, Ø., Rutle, A.: DPF Editor: A Multi-Layer Diagrammatic (Meta)Modelling Environment. In: Proceedings of SPLST 2011: 12th Symposium on Programming Languages and Software (2011)
16. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The Generic Modeling Environment. In: Proceedings of WISP 2001: Workshop on Intelligent Signal Processing, vol. 17, pp. 82–83. ACM (2001), http://www.isis.vanderbilt.edu/sites/default/files/GME2000Overview.pdf
17. Lengyel, L., Levendovszky, T., Charaf, H.: Constraint Validation Support in Visual Model Transformation Systems. Acta Cybernetica 17(2), 339–357 (2005)
18. Object Management Group: Meta-Object Facility Specification (2006), http://www.omg.org/spec/MOF/2.0/
19. Object Management Group: Object Constraint Language Specification (2010), http://www.omg.org/spec/OCL/2.2/
20. Object Management Group: Unified Modeling Language Specification (2010), http://www.omg.org/spec/UML/2.3/
21. Rabbi, F., Mashiyat, A.S., MacCaull, W.: Model checking workflow monitors and its application to a pain management process. In: Proceedings of FHIES 2011: 1st International Symposium on Foundations of Health Information Engineering and Systems, pp. 110–127 (2011), http://www.iist.unu.edu/ICTAC/FHIES2011/Files/fhies2011_8_17.pdf
22. Reinhard, T., Seybold, C., Meier, S., Glinz, M., Merlo-Schett, N.: Human-Friendly Line Routing for Hierarchical Diagrams. In: Proceedings of ASE 2006: 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 273–276. IEEE Computer Society (2006)
23. Rutle, A.: Diagram Predicate Framework: A Formal Approach to MDE. Ph.D. thesis, Department of Informatics, University of Bergen, Norway (2010)
24. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A Diagrammatic Formalisation of MOF-Based Modelling Languages. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. LNBIP, vol. 33, pp. 37–56. Springer, Heidelberg (2009)
25. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A Formalisation of Constraint-Aware Model Transformations. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 13–28. Springer, Heidelberg (2010)
26. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2008)
27. TOPCASED: Project Web Site, http://www.topcased.org