Proceedings of the Estonian Academy of Sciences,
2013, **62**, 1, 3–15
doi: 10.3176/proc.2013.1.02
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

DPF Workbench: a multi-level language workbench for MDE

Yngve Lamo[a*], Xiaoliang Wang[a], Florian Mantz[a], Øyvind Bech[a], Anders Sandven[a], and Adrian Rutle[b]

[a] Bergen University College, P.O. Box 7030, Nygårdsgaten 112, N-5020 Bergen, Norway
[b] Aalesund University College, P.O. Box 1517, 6025 Aalesund, Norway

Received 31 August 2011, revised 11 December 2012, accepted 17 December 2012, available online 20 February 2013

Abstract. This paper presents the DPF Workbench, a language workbench for (meta)modelling and code generation. The DPF Workbench includes a graphical specification editor for the Diagram Predicate Framework (DPF), which provides a graph-based formalization of (meta)modelling and model transformation. The tool offers functionality for fully diagrammatic specifications of domain-specific modelling languages. Moreover, the DPF Workbench supports the development of metamodelling hierarchies with an arbitrary number of metalevels; i.e. each model at a metalevel can be used as a metamodel for the metalevel below. The DPF Workbench facilitates the generation of domain-specific diagrammatic editors out of these metamodels. The conformance relations between adjacent metalevels are checked using typing morphisms and validation of diagrammatic constraints. In addition, the DPF Workbench provides a signature editor for the definition of software constraints and their corresponding validators. The code generator is a newly added component that facilitates the generation of software from models defined in the DPF Workbench. The features of the DPF Workbench are illustrated by a running example presenting a metamodelling hierarchy for business process modelling and sketching how these models can be transformed to programs by the code generation facility.

Key words: model-driven engineering, diagram predicate framework, language workbench, diagrammatic modelling, metamodelling.

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) [26]. To fully unfold the potential of MDE, models are specified by Domain-Specific Modelling Languages (DSMLs) that are tailored to a specific domain of concern [13]. The DSMLs are modelling languages where the language primitives consist of domain concepts. It is common practice to specify these domain concepts by a graph-based metamodel while the constraints are specified by a text-based language such as the Object Constraint Language (OCL) [25]. This mixture of text-based and graph-based languages is an obstacle for employing MDE, especially regarding model transformation [33] and synchronization of graphical models with their textual constraints [30]. 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.

The availability of tools that facilitate the design and implementation of DSMLs is another important factor for the acceptance and adoption of MDE. DSMLs are specified by metamodels, hence it is necessary to be able to automatically create modelling tools from these metamodels. To be useful, DSMLs are required to be intuitive enough for domain experts, while they are formal enough to enable sound model transformations and code generation. Therefore, we propose a formal, diagrammatic approach to (meta)modelling and generation of DSMLs.

* Corresponding author, yla@hib.no

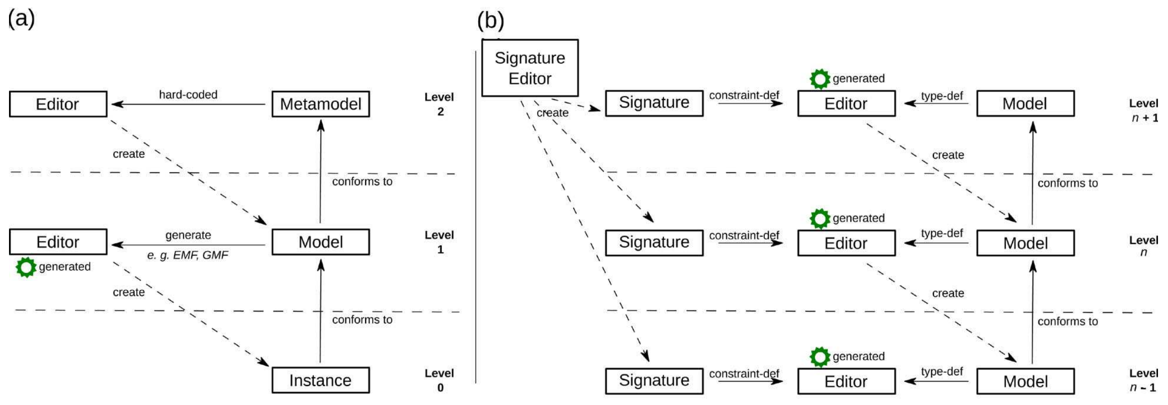


Fig. 1. A simplified view of (a) the EMF metamodelling hierarchy and (b) a generic metamodelling hierarchy as implemented in the DPF Workbench.

An industrial standard to describe DSMLs is the Meta-Object Facility (MOF) [24] 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) [35]. 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; 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. Note that the EMF is a modelling framework with code generation facilities for defining structural data models, but the functionality for creating diagrammatic DSMLs is not in the scope of the EMF.

The two-level metamodelling approach has several limitations (see [1,15,28] for a comprehensive argumentation). The lack of multi-level metamodelling support forces DSML designers to introduce type–instance relations in the metamodel. This leads to a mixture of domain concepts with language concepts at the same modelling level. The approach in this paper tackles this issue by introducing a multi-level metamodelling tool. That is, a tool for developing metamodelling hierarchies with an arbitrary number of metalevels where each model at a metalevel can be used as a metamodel for the metalevel below (see Fig. 1b) is introduced.

The automatic generation of software from models is one of the fundamental ideas of MDE. It enhances productivity, code quality, consistency, etc. [17]. By combining code generation techniques with DSMLs, it is possible to separate domain concerns from implementation details. Traditionally, code generation is done for two meta-level modelling hierarchies. In this paper we propose a code generation approach for multi-level metamodelling hierarchies. In this way we can construct code generators for any level of abstraction, which makes it possible to create high-level prototypes of the system to test design choices in an early phase of the software development process.

Language workbench is a concept that has gained popularity in the last years. A language workbench consists of an environment for creating DSML/DSLs and corresponding tools [13]. A workbench should provide an IDE-like environment for creating DSML/DSLs, and in addition, it should generate tooling support for the specified language that facilitates code generation, model transformation, model versioning, etc. DSMLs are usually specified by metamodels that are defined as instances of a general purpose modelling language.

When working with language workbenches, the development process is divided into two phases. First, the DSML is created along with relevant tools, such as editors and code generators (see Fig. 2). This activity should be performed by experienced developers or language designers in collaboration with domain experts, ensuring that the tooling for the language is tailored to the user's needs. Second, developers use the DSML and the corresponding tools to create software for the application domain. To enhance the communication it is important to use a development language that is understood by both the developers and the domain experts.

This paper presents the DPF Workbench, a language workbench for (meta)modelling and code generation. The DPF Workbench is an implementation of the techniques and methodologies developed in the Diagram Predicate Framework (DPF) [6,32,33], which provides a formalization of (meta)modelling and model transformations. The DPF is an extension of the generalized sketches framework [9,10], based on category theory [5] and graph transformations [12]. The DPF Workbench supports the 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. A DPF model consists of a graph-based structure together with a set of diagrammatic constraints. The conformance of models to metamodels is formalized and checked in the DPF Workbench by validating both typing and diagrammatic constraints.

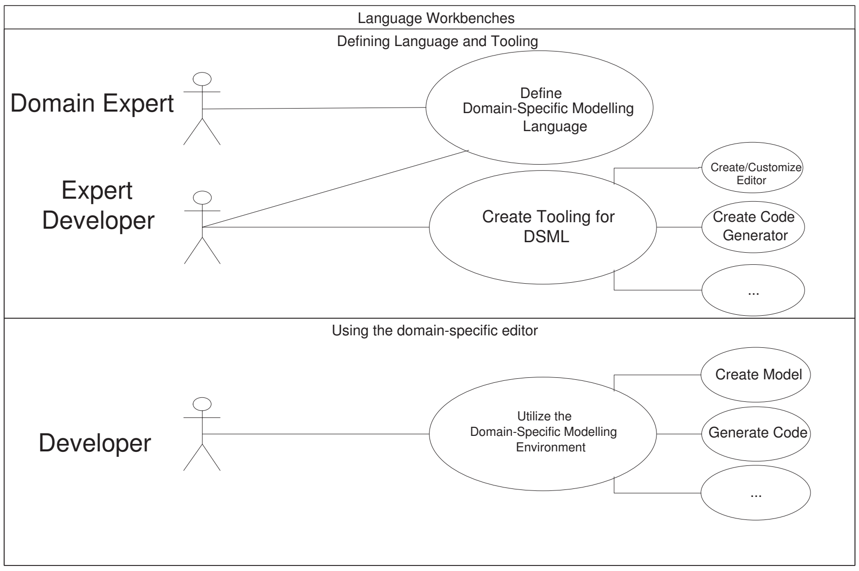


Fig. 2. Intended use of language workbenches.

The DPF specification editor was presented for the first time in [20]. Later, in [21], the DPF specification editor was extended with an editor for creating diagrammatic signatures facilitating the definition of user-defined predicates. In this paper we combine the functionality from the specification and signature editors with the newly added code generation functionality, which is an important step on the way to make a complete MDE workbench. Moreover, the comparison to related approaches is more elaborated in this version than in earlier works.

The remainder of the paper is organized as follows. Section 2 introduces some basic concepts from the DPF. Section 3 gives a brief overview of the tool architecture. Section 4 demonstrates the metamodelling functionality of the tool by giving an example of a metamodelling scenario. Section 5 introduces the code generation facilities and generates code for the example presented in Section 4. Section 6 compares the DPF Workbench with related tools, and Section 7 concludes the paper and outlines future research directions and possible extensions of the DPF Workbench.

2. DIAGRAM PREDICATE FRAMEWORK

We will now give a brief introduction to the DPF, for details please check the papers on the foundation of the DPF, e.g. [30,32]. In the 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* $C^{\mathfrak{S}}$. 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 visualization, and a semantic interpretation (see Table 1). A constraint is given by a predicate together with the subgraph of the specification's underlying graph that is affected by the predicate.

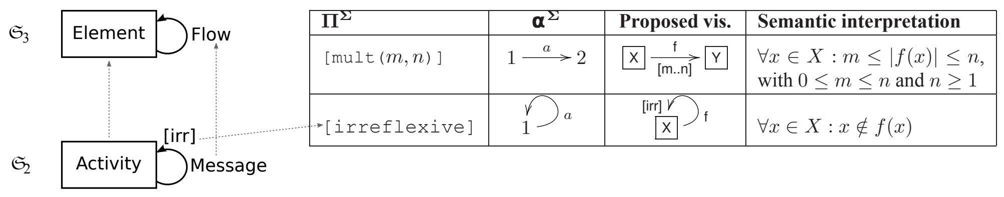
For instance, Fig. 3 shows a specification $\mathfrak{S}_2$ that is compliant with the requirement "activities cannot send messages to themselves". In $\mathfrak{S}_2$, this requirement is forced by the atomic constraint $(\texttt{[irreflexive]}, \delta)$ on the arrow Message. Note that δ is a graph homomorphism $\delta : \alpha^{\Sigma}(\texttt{[irreflexive]}) \rightarrow \mathfrak{S}_2$ specifying which part of $\mathfrak{S}_2$ is affected by the `[irreflexive]` predicate.

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. 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 $f : X \rightarrow \wp(Y)$. 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 $f : X \rightarrow \wp(Y)$, $g : Y \rightarrow \wp(Z)$ 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). An instance (I, ι) of $\mathfrak{S} = (S, C^{\mathfrak{S}})$ is a graph I together with a graph homomorphism $\iota : I \rightarrow S$ 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 [30]. 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.

Table 1. The signature Σ used in the metamodelling example

Π^{Σ}	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
`[mult(`m,n`)]`	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[\text{[m..n]}]{f} \boxed{Y}$	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[irreflexive]`	$\begin{tikzcd} 1 \arrow[loop right, "f"] \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[loop right, "f", "\text{[irr]}"'] \end{tikzcd}$	$\forall x \in X : x \notin f(x)$
`[injective]`	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[\text{[inj]}]{f} \boxed{Y}$	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[nand]`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"'] & 2 \\ 3 \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[r, "f"] \arrow[d, "g"'] \arrow[r, phantom, bend right, "\text{[nand]}"'] & \boxed{Y} \\ \boxed{Z} \end{tikzcd}$	$\forall x \in X :$ $f(x) = \emptyset \vee g(x) = \emptyset$
`[surjective]`	$1 \xrightarrow{f} 2$	$\boxed{X} \xrightarrow[\text{[surj]}]{f} \boxed{Y}$	$f(X) = Y$
`[jointly-surjective_2]`	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \\ & 3 \arrow[u, "g"'] \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[r, "f"] \arrow[r, phantom, bend right, "\text{[js]}"'] & \boxed{Y} \\ & \boxed{Z} \arrow[u, "g"'] \end{tikzcd}$	$f(X) \cup g(Z) = Y$
`[xor]`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"'] & 2 \\ 3 \end{tikzcd}$	$\begin{tikzcd} \boxed{X} \arrow[r, "f"] \arrow[d, "g"'] \arrow[r, phantom, bend right, "\text{[xor]}"'] & \boxed{Y} \\ \boxed{Z} \end{tikzcd}$	$\forall x \in X :$ $(f(x) = \emptyset \vee g(x) = \emptyset)$ and $(f(x) \neq \emptyset \vee g(x) \neq \emptyset)$



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

Fig. 3. Specifications $\mathfrak{S}_2$ and $\mathfrak{S}_3$ and the signature Σ.

In the 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 $\iota_i : S_i \longrightarrow S_{i+1}$, 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 $\iota_i : S_i \longrightarrow S_{i+1}$ such that (S_i, ι_i) is an instance of $\mathfrak{S}_{i+1}$; i.e. such that ι_i satisfies the atomic constraints $C^{\mathfrak{S}_{i+1}}$.

For instance, Fig. 3 shows a specification $\mathfrak{S}_2$ that conforms to a specification $\mathfrak{S}_3$. That is, there exists a typing morphism $\iota_2 : S_2 \longrightarrow S_3$ such that (S_2, ι_2) is an 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 an instance of $\mathfrak{S}_3$ as long as there exists a typing morphism $\iota_2 : S_2 \rightarrow S_3$. However, Fig. 4 shows two graphs, both typed by the specification $\mathfrak{S}_2$, but only Fig. 4a is an instance of $\mathfrak{S}_2$, since the graph in Fig. 4b violates the $(\texttt{[irreflexive]}, \delta)$ constraint on the arrow Message.

To ensure correct typing, the DPF Workbench only allows creation of elements that are typed by the elements in the metamodel. Hence in the example above, when we create the nodes $\mathsf{Activity}_1$ and $\mathsf{Activity}_2$, we use the tooling palette of the DPF Workbench to choose that the meta-type of these two elements is the node Activity in $\mathfrak{S}_2$. Moreover, when we create the edge m_1 of type Message, the DPF Workbench checks if the corresponding types of the source and the target of m_1 are correct before we are allowed to create it. This means that the source and target of m_1 must be typed by the source and target of Message, respectively. Thus, the DPF Workbench actually constructs a graph homomorphism when new elements are created. This graph homomorphism is stored with the model, which makes later type checking a trivial task.

To ensure constraint satisfaction, for each constraint c on a subgraph H of the metamodel, the DPF Workbench checks if the part of the model that is typed by H fulfills c. The actual check is done by running a Java (or OCL) validator that checks whether the constraint is satisfied by the part of the model that is typed by H. In the example above, there is an irreflexivity constraint $(\texttt{[irreflexive]}, \delta)$ on the meta-type Message, and the edges m_1 and m_2 are typed by Message. To check whether the constraint is satisfied by the graph in Fig. 4a, the DPF Workbench checks whether the common part (the pullback) of δ and ι satisfies the constraint validator of `[irreflexive]`, where ι is the typing graph homomorphism that was constructed when the model was created. The constraint validator of `[irreflexive]` simply checks whether there is a loop in the graph.

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

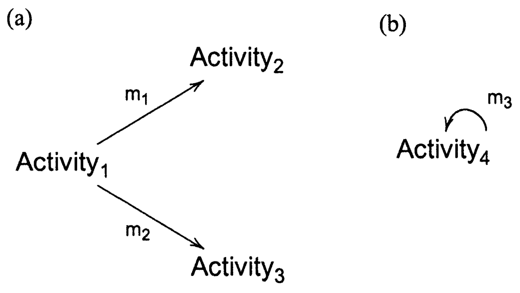


Fig. 4. (a) An instance of $\mathfrak{S}_2$ and (b) a graph that violates the irreflexivity constraint of $\mathfrak{S}_2$.

3. TOOL ARCHITECTURE

The DPF Workbench was developed in Java as a plug-in for Eclipse [11]. Eclipse follows a cross-platform architecture that is well suited for tool integration since it implements the Open Services Gateway initiative (OSGi) framework. Moreover, it has an ecosystem around the basic Eclipse 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 practice and is also employed in commercial products such as the Rational Software Architect (RSA) [18] as well as in open-source products such as the modelling tool TOPCASED [38]. As an Eclipse plug-in the DPF Workbench can easily be integrated into and used together with such tools.

Figure 5 illustrates that the DPF Workbench consists of three main components, which are built on top of three auxiliary components. The auxiliary component “DPF Core” provides access to the core features of the tool: these are the facilities to create, store, and validate DPF specifications. This part uses EMF for data storage. Thus 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 was chosen for data storage because 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. The component “DPF Core” is extended by the auxiliary component “DPF Diagram”, which also contains an EMF metamodel that stores additional information that is used to visualize models. Such models store e.g. the position of elements and information concerning their visualization. The component “DPF Diagram” depends on the Graphical Editing Framework (GEF) [16]. The GEF provides functionalities to create rich graphical

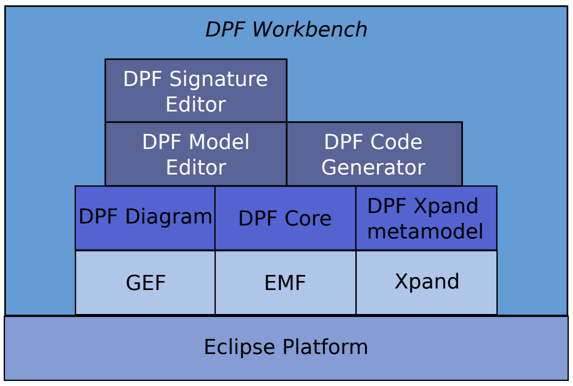


Fig. 5. The main component architecture of the DPF Workbench plug-in packages.

editors and views for the Eclipse platform following a Model–View–Controller (MVC) architecture. The last auxiliary component implements an extension to the Xpand generator (see Section 5).

The three main components are the "DPF Model Editor", the "DPF Signature Editor", and the "DPF Code Generator". The first two of these components implement the modelling functionality of the tool. The "DPF Model Editor" is the component that allows creation and modification of DPF specifications. It uses "DPF Core" and "DPF Diagram" and also implements the view part of the GEF's MVC architecture. Special arrow-routing and display functions have been developed for showing DPF's special kinds of predicates. The "DPF Signature Editor" extends the functionality of the "DPF Model Editor" by providing an editor for building user-defined, reusable predicate signatures [21]. It mainly relies on the functionality provided by the "DPF Model Editor" component but it also uses some of the functionalities provided by the "DPF Core". The third component, "DPF Code Generator", builds on top of the "DPF Core" and "DPF Xpand metamodel" and makes code generation facilities available for users via a wizard.

4. A METAMODELLING EXAMPLE

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

The DPF Workbench 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. The details of how signatures are created are given in [21].

We start the metamodelling process by configuring the tool with the DPF Workbench's default metamodel $\mathfrak{S}_4$, consisting of Node and Arrow, which 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 business process 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, which 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.

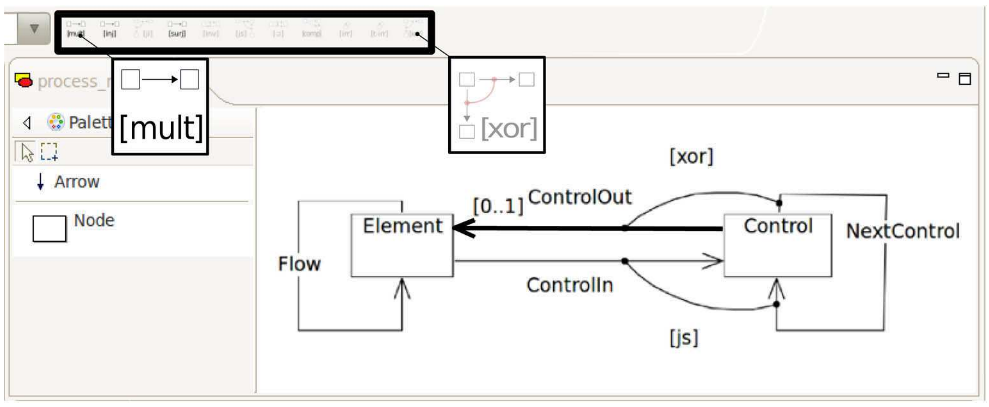


Fig. 6. DPF Workbench configured with the default metamodel consisting of Node and Arrow, and the signature Σ from Table 1, indicated with a bold black rectangle; showing also the specification $\mathfrak{S}_3$ under construction. Note that the bold black arrow ControlOut is selected, therefore the predicates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

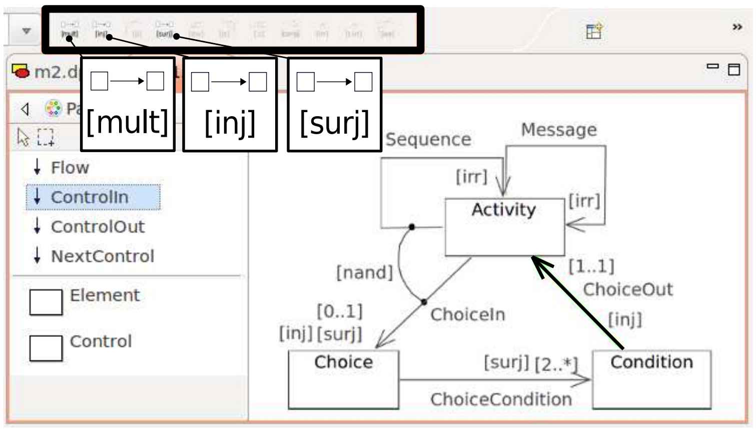


Fig. 7. DPF Workbench configured with the specification $\mathfrak{S}_3$ from Fig. 6 as metamodel, and the signature Σ from Table 1 indicated with a bold rectangle; also the specification $\mathfrak{S}_2$ under construction is shown.

Now we will illustrate how an editor can be generated 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 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 still use the same signature from Table 1 with this new 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 *messages* 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 to 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 a maximum of one *condition* connected to it.
9. An *activity* cannot send messages to itself.
10. An *activity* cannot be sequenced to itself.

We now explain how some of the requirements above are specified in $\mathfrak{S}_2$. 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. Requirement 5 is specified by adding the constraint `[nand]` on the arrows Sequence and Choice. Requirement 6 is specified by adding the constraints `[injective]` and `[surjective]` on ChoiceIn. Requirements 9 and 10 are specified by adding the constraint `[irreflexive]` on Message and Sequence, respectively.

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 Workbench 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 the 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 that 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 by both". This violation will be indicated in the tool by a message (or a tip) in the status bar.

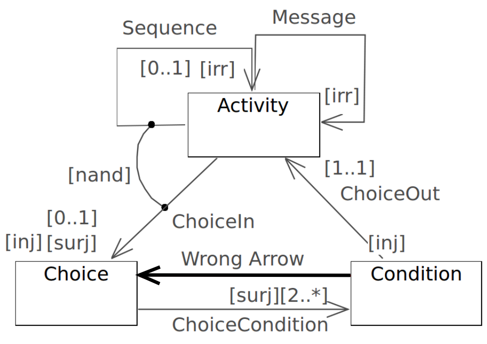


Fig. 8. A specification violating `[xor]` constraint on the arrows ControlOut and NextControl in $\mathfrak{S}_3$.

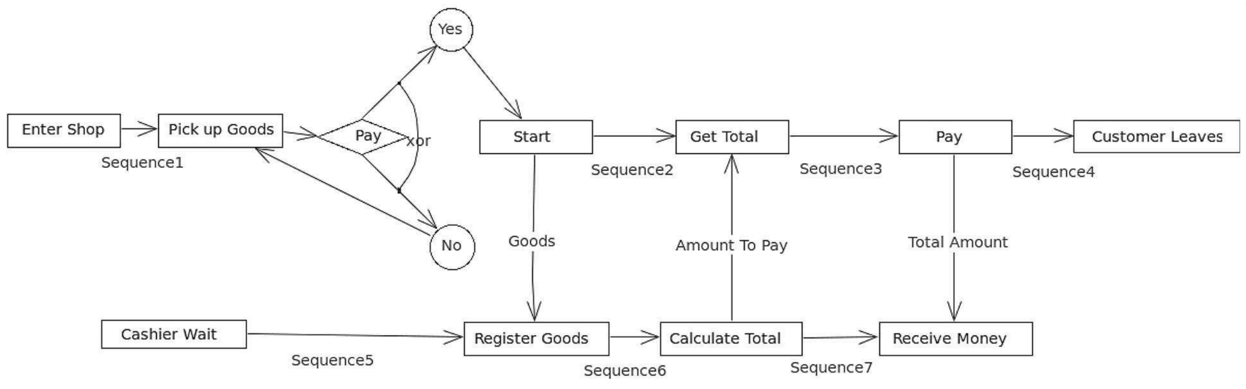


Fig. 9. A sample business process model $\mathfrak{S}_1$ for purchasing, specified by the DPF Workbench configured with the specification $\mathfrak{S}_2$ as metamodel.

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. 9. Note that the tool palette in Fig. 9 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 Condition and Choice are visualized as diamonds and circles, respectively.

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 example we stop at this level, and will eventually generate code from $\mathfrak{S}_1$ as illustrated in the next section.

5. CODE GENERATION

We will now illustrate the newly added code generation facility of the DPF Workbench. Code generation is the process of automatically creating programming code from software models. The models specify the software at a high level of abstraction – independent of the implementation details, whereas the code generator creates the executable source code from the models. Code generation is a common practice in today's software development and has become a built-in functionality in modern IDEs. Large middleware platforms, such as Java EE and Spring with their extensive need for configuration files, have shown the advantages of code generation. These configuration files, often written in XML, are tedious and time consuming to maintain. The focus in this paper will not be on generating specific configuration files for an arbitrary framework, instead, it will be on a general solution for creating code from domain-specific models.

In language workbenches, code generation is the usual way to construct tool support for DSMLs. In MDE, code generation is viewed as a special form of model transformations, more specifically model-to-text transformations [23]. Hence the actual code generation is performed by executing a sequence of transformation rules resulting in the executable software code. A transformation rule could be an expression in a template, or defined in a general purpose programming language. In template-based code generation, templates are used to define how model elements are transformed to source code based on their types. The metamodel specifies the types of the DSML and the template describes how instances of these types should be transformed to code. Xpand is a popular code generation template language used with Eclipse. Xpand supports code generation from e.g. EMF or XSD models. In this paper, we adapt Xpand so that it can be used for code generation from DPF models at any meta-level in the DPF hierarchy. A corresponding Xpand template is created for each DSML, which can be used to transform models created by the DSML to source code. Note that since EMF only allows modelling hierarchies with two meta-levels, Xpand is usually bound to code generation for these two levels.

Now we illustrate the code generation facility of the DPF Workbench by creating a template for the metamodelling hierarchy defined in the previous section. For more details of the template-based code generation facility in the DPF Workbench see [34].

To generate code, a template has to be created that uses a DPF specification as metamodel. A default, ready-to-use template can be created by the new wizard "DPF Generator Project". Listing 1 shows an excerpt of a template that generates Java code for models conforming to the specification in Fig. 7. The template uses the usual Xpand syntax. The Xpand generator framework is extended to deal with DPF specifications at any meta-level. The template language support standard control structures like `FOREACH` loops and supports also sub-templates that are defined by `DEFINE` statements and called by `EXPAND` statements.

Listing 1. An example template

```
«DEFINE Main FOR dpf::Specification»
    «EXPAND graph FOR this.graph»
«ENDDEFINE»

«DEFINE graph FOR dpf::Graph»
    «EXPAND activity FOREACH this.getActivitys()»
    «EXPAND choice FOREACH this.getChoices()»
    «EXPAND condition FOREACH this.getConditions()»
«ENDDEFINE»

«DEFINE activity FOR dpf::Activity»
    «FILE this.name.replaceAll(" ","").toFirstUpper() + ".java"»
      public class «this.name.replaceAll(" ","").toFirstUpper()»
                    implements Activity{
      «FOREACH getASequences() AS e»
          «FILE e.name.replaceAll(" ","").toFirstUpper() + ".java"»
          import java.util.*;
          public class «e.name.replaceAll(" ","").toFirstUpper()»
                        implements Sequence{
            public «e.source.name.replaceAll(" ","").toFirstUpper()» source;
            public «e.target.name.replaceAll(" ","").toFirstUpper()» target;
          }
          «ENDFILE»
          «e.name.replaceAll(" ","").toFirstUpper()» next;
      «ENDFOREACH»
```

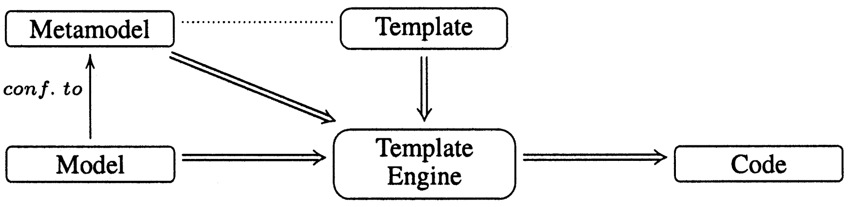


Fig. 10. Code generation overview.

Listing 2. Example 2 generated Java Class

```
public class EnterShop implements Activity {
    Sequence1 next;
}
```

To run code generation on a certain model, the model must be linked to the template and the metamodel. The template engine can then iterate over the model elements and create code following the template (see Fig. 10). Since we support Xpand's workflow approach this can be done by editing a workflow file that is also generated by the DPF generator project wizard. After inserting the path to the model, the file can be executed by "Run As → MWE Workflow". Furthermore, the workflow file can be augmented with post-processors, such as pretty printers, to automatically format the generated code. The Java classes in Listings 2 and 3 are example classes generated by the template engine from the model in Fig. 9 as input. Because of space limitations in the paper, we kept the template file simple intentionally.

Listing 3. Example 1 generated Java Class

```
import java.util.*;
public class Sequence1 implements Sequence {
    public EnterShop source;
    public PickupGoods target;
}
```

6. RELATED WORK

There is an abundance of visual modelling tools available, both as open-source software and as 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 [22]. The 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.

A Tool for Multi-formalism and Meta-Modelling (AToM^3) is a tool for multi-paradigm modelling [2,8]. 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) [14] 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. 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 visualization is customizable 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. The 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 it only runs on the Microsoft Windows platform.

MetaEdit [37] is a commercial tool offering (meta)modelling and code generation functionality. The tool clearly separates (meta)models from their diagrammatic visualization; it also offers functionality for code generation. The potential to customize the visual presentation of the modelling elements is better than in the other compared tools. MetaEdit uses a two-layer modelling hierarchy. The main limitation of the tool is the treatment of constraints: it only supports a set of predefined constraints over binary relations. There is no way that the user can define new domain-specific constraints. MetaEdit is a stand-alone proprietary application that runs on the Windows, Linux, and Mac platforms, it also provides plug-ins both for Eclipse and Visual Studio.

The metaDepth [7] 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 formalization of metaDepth and deep metamodelling in general.

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

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

Tool	Layers	Code Generation	Constraint Language	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	✓
MetaEdit+	2	✓	Predefined	Java VM	✓
metaDepth	∞	✓	EVL	Java VM	
DPF Workbench	∞	✓	Validators	Java VM	✓

7. CONCLUSION AND FUTURE WORK

This paper extends [20] with code generation facilities for the DPF Workbench, whereas in [21] the DPF editor was extended with functionality to define signature editors. The DPF Workbench is an open-source project and can be downloaded from dpf.hib.no. It is developed in Java and runs as a plug-in on the Eclipse platform. It supports fully diagrammatic metamodelling as proposed by the DPF and it has also a built-in code generation facility that is based on Xpand templates. The templates define what to generate from model elements based on their types in the metamodel, and the template engine takes a model conforming to the metamodel as input and generates code out of it following the instructions in the template. The functionality of the tool has been illustrated by specifying a metamodelling hierarchy for business process modelling and generating Java code for it. We outline 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 also showed how models created in the tool can be used as metamodels. The authors are not aware of other EMF-based tools that facilitate multi-level metamodelling, especially with code generation support for models at any level of the hierarchy. This functionality could be used for testing design choices early in the development phase by generating prototypes.

The tool was used in a graduate course in MDE at Bergen University College in Spring 2012. The students participated in a field experiment designed for testing the DPF Workbench. This experiment gave positive user feedback from participants external to the development project. The modelling part was quickly learned by the students, they commented that it was easy to learn since the tool basically has only two primitives, node and edge. The students needed more training to get used to the DPF constraints, and proposed directions to improve the tool, especially concerning the visual syntax.

The DPF Workbench was recently used in an industrial case study in model-driven development of web services [19]. The study shows that it is possible to use the tool for modelling web services and to use the code generator to deploy the web services.

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

- **Configurable concrete syntax.** As the system exists today, all diagram (nodes, arrows, and constraints) visualizations are hardcoded in the editor code. A desirable extension would be to make visualization models 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 great usability gain to be capitalized on in this matter. Today's editor contains a simple routing algorithm, based on the GEF's `ShortestPathConnectionRouter` class. The problem of finding routing algorithms that produce easy-to-read output is a focus of continuous research [27], and this problem applied to the DPF Workbench can probably be turned into a separate research task.
- **(Meta)model evolution.** Metamodels of DSMLs evolve during their life cycles; it is important that models and other artefacts are changed correspondingly. Some preliminary work has been done in this direction in [36]; in our future work we will build on this work and provide an implementation for the DPF Workbench.
- **Model versioning.** Usually development environments are rather spread and models are developed concurrently by different groups from models different locations. Tool support for calculating model differences and merging the changes will increase the usability of the DPF Workbench. The theoretical work is already established in [29], and an implementation of this is planned in the future.

- **Behavioural modelling.** Traditionally MDE has focused on structural modelling, but it is an emerging trend to also use MDE techniques to develop behavioural models. We have already extended the formal foundation of the DPF to support behavioural modelling in [31] and the DPF Workbench has been used to create a DSML for specifying health care workflows. Currently the code generation facility described in this paper is used to generate DiVinE [4] code from these behavioural models, which will then be used to check properties against the models.

In addition to these areas, development to utilize the core functionality of the 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. and Kühne, T. Rearchitecting the UML infrastructure. *TOMACS*, 2002, **12**(4), 290–321.
2. AToM^3: A Tool for Multi-formalism and Meta-Modelling. *Project Web Site*. http://atom3.cs.mcgill.ca/ (accessed 10.12.2012).
3. Baar, T. Correctly defined concrete syntax for visual modeling languages. In *MoDELS 2006* (Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., eds). LNCS, Vol. 4199. Springer, 2006, 111–125.
4. Barnat, J., Brim, L., Češka, M., and Ročkai, P. DiVinE: parallel distributed model checker (tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*. IEEE, 2010, 4–7.
5. Barr, M. and Wells, C. *Category Theory for Computing Science*. 3rd edn. Les Publications CRM, Montreal, 1999.
6. Bergen University College and University of Bergen. *Diagram Predicate Framework Web Site*. http://dpf.hib.no/ (accessed 10.12.2012).
7. De Lara, J. and Guerra, E. Deep meta-modelling with METADEPTH. In *TOOLS 2010* (Vitek, J., ed.). LNCS, Vol. 6141. Springer, 2010, 1–20.
8. De Lara, J. and Vangheluwe, H. Using AToM^3 as a Meta-CASE Tool. In *ICEIS 2002*. April 2002, 642–649.
9. Diskin, Z. Mathematics of generic specifications for model management, I and II. In *Encyclopedia of Database Technologies and Applications*. Information Science Reference, 2005, 351–365.
10. Diskin, Z., Kadish, B., Piessens, F., and Johnson, M. Universal arrow foundations for visual modeling. In *Diagrams 2000* (Anderson, M., Cheng, P., and Haarslev, V., eds). LNCS, Vol. 1889. Springer, 2000, 345–360.
11. Eclipse Platform. *Project Web Site*. http://www.eclipse.org (accessed 10.12.2012).
12. Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.
13. Fowler, M. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.
14. GME: Generic Modeling Environment. *Project Web Site*. http://www.isis.vanderbilt.edu/Projects/gme/ (accessed 10.12.2012).
15. Gonzalez-Perez, C. and Henderson-Sellers, B. *Metamodelling for Software Engineering*. Wiley, 2008.
16. Graphical Editing Framework. *Project Web Site*. http://www.eclipse.org/gef/ (accessed 10.12.2012).
17. Herrington, J. *Code Generation in Action*. Revised edition. Manning Publications, July 2003.
18. IBM. *Rational Software Architect*. http://www-01.ibm.com/software/awdtools/architect/swarchitect/ (accessed 10.12.2012).
19. Kalakata, S. Formalisation of Simple MethOd Declaration Language (SMODL) by DPF. Master's thesis, Department of Computer Engineering, Bergen University College, Norway, November 2012.
20. Lamo, Y., Wang, X., Mantz, F., Bech, Ø., and Rutle, A. DPF editor: a multi-layer diagrammatic (meta)modelling environment. In *SPLST 2011* (Penjam, J., ed.). TUT Press, Tallinn, October 2011, 55–65.
21. Lamo, Y., Wang, X., Mantz, F., MacCaull, W., and Rutle, A. DPF workbench: a diagrammatic multilayer domain specific (meta-)modelling environment. In *Computer and Information Science* (Roger, L., ed.). Studies in Computer Intelligence, Vol. 429. Springer, 2012, 37–52.
22. Lengyel, L., Levendovszky, T., and Charaf, H. Constraint validation support in visual model transformation systems. *Acta Cybernetica*, 2005, **17**(2), 339–357.
23. Mens, T. and Van Gorp, P. A taxonomy of model transformation. *ENTCS*, 2006, **152**, 125–142.
24. Object Management Group. *Meta-Object Facility Specification*, January 2006. http://www.omg.org/spec/MOF/2.0/ (accessed 10.12.2012).
25. Object Management Group. *Object Constraint Language Specification*, February 2010. http://www.omg.org/spec/OCL/2.2/ (accessed 10.12.2012).
26. Object Management Group. *Unified Modeling Language Specification*, May 2010. http://www.omg.org/spec/UML/2.3/ (accessed 10.12.2012).
27. Reinhard, T., Seybold, C., Meier, S., Glinz, M., and Merlo-Schett, N. Human-friendly line routing for hierarchical diagrams. In *ASE 2006*. IEEE Computer Society, 2006, 273–276.
28. Rossini, A., de Lara, J., Guerra, E., Rutle, A., and Lamo, Y. A graph transformation-based semantics for deep metamodelling. In *AGTIVE 2011* (Schürr, A., Varró, D., and Varró, G., eds). LNCS, Vol. 7233. Springer, 2011, 19–34.
29. Rossini, A., Rutle, A., Lamo, Y., and Wolter, U. A formalisation of the copy-modify-merge approach to version control in MDE. *JLAP*, 2010, **79**(7), 636–658.
30. Rutle, A. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

31. Rutle, A., MacCaull, W., Wang, H., and Lamo, Y. A metamodelling approach to behavioural modelling. In *BM-FA '12*. ACM, 2012, 5:1–5:10.
32. Rutle, A., Rossini, A., Lamo, Y., and Wolter, U. A diagrammatic formalisation of MOF-based modelling languages. In *TOOLS 2009* (Oriol, M., Meyer, B., Aalst, W. et al., eds). LNBIP, Vol. 33. Springer, 2009, 37–56.
33. Rutle, A., Rossini, A., Lamo, Y., and Wolter, U. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 2012, **81**(4), 422–457.
34. Sandven, A. Metamodel based Code Generation in DPF Editor. Master's thesis, Department of Informatics, University of Bergen, Norway, April 2012.
35. Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. *EMF: Eclipse Modeling Framework 2.0.* 2nd edn. Addison-Wesley Professional, 2008.
36. Taentzer, G., Mantz, F., and Lamo, Y. Co-transformation of graphs and type graphs with application to model co-evolution. In *ICGT 2012* (Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G., eds). LNCS, Vol. 7562. Springer, 2012, 326–340.
37. The Entity MetaEdit+ Workbench. *Project Web Site*. http://www.metacase.com/mep/ (accessed 10.12.2012).
38. TOPCASED. *Project Web Site*. http://www.topcased.org (accessed 10.12.2012).

DPF-tööriistakast: mitmetasandilised keeleprotsessorid mudel-orienteeritud projekteerimiseks

Yngve Lamo, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven ja Adrian Rutle

On tutvustatud DPF-tööriistakasti ja (meta)modelleerimiseks ning koodi genereerimiseks sobivaid keeleprotsessoreid. DPF-tööriistakasti kuulub graafiline spetsifikatsioonide redaktor DPF (Diagram Predicate Framework), mis võimaldab (meta)mudeleid ja nende teisendusi graafiliselt formaliseerida. Redaktori funktsionaalsus võimaldab probleem-orienteeritud keelte täielikku spetsifitseerimist diagrammide abil. Lisaks toetab DPF-tööriistakast metamudelite hierarhiate kirjeldamist suvaliste hierarhiatasandite jaoks, st iga metatasandi mudelit saab allpool asetseval tasandil metamudelina kasutada. DPF-tööriistakast hõlbustab valdkonnaspetsiifiliste skeemiredaktorite genereerimist metamudelitest. Naabertasandite metamudelite konformsus tagatakse tüübimorfismide abil ja teatud diagrammatiliste kitsenduste valideerimise teel. Lisaks on DPF-tööriistakastis signatuuride redaktor tarkvarakitsenduste ja vastavate validaatorite defineerimiseks. Koodigeneraator on hiljuti lisandunud komponent, mis võimaldab mudelitest tarkvara genereerida. DPF-tööriistakasti võimalusi on artiklis näidatud praktilise näite varal, esitades äriprotsesside metamudelite hierarhia ja visandades protsessi, kuidas saaks neid mudeleid koodigeneraatori abil programmideks teisendada.