

DropsBox: The Dresden Open Software Toolbox

Domain-Specific Modelling Tools beyond Metamodels and Transformations

Uwe Aßmann, Andreas Bartho, Christoff Bürger, Sebastian Cech, Birgit Demuth, Florian Heidenreich, Jendrik Johannes, Sven Karol, Jan Polowinski, Jan Reimann, Julia Schroeter, Mirko Seifert, Michael Thiele, Christian Wende, Claas Wilke

Institut für Software- und Multimediatechnik, Technische Universität Dresden, Germany
DevBoost GmbH, Berlin, Germany
e-mail: `firstname.lastname@tu-dresden.de`

Received: date / Revised version: date

Abstract The Dresden Open Software Toolbox (DropsBox) is a software modelling toolbox consisting of a set of open source tools developed by the Software Technology Group at TU Dresden. The DropsBox is built on top of the Eclipse Platform and the Eclipse Modeling Framework. The DropsBox contributes to the development and application of domain-specific languages (DSLs) in model-driven software development. It can be customised by tool and language developers to support various activities of a DSL's life cycle ranging from language design to language application and evolution. In this paper, we provide an overview of the DSL life cycle, the DropsBox tools, and their interaction on a common example. Furthermore, we discuss our experiences in developing and integrating tools for DropsBox in an academic environment.

Key words Domain-Specific Modelling Environment, Domain-Specific Language, Language Life Cycle, Modelling Tool, MDSD, EMF

1 Introduction

Domain-specific modelling (DSM) and model-driven software development (MDSD) propose the creation and exploitation of models, i.e., abstract representations of the knowledge about a particular domain, in the design and implementation of software systems. Models are typically built by using custom languages—called DSLs—that match the concepts found in a particular domain. This is meant to ease the representation, analysis, or processing of domain knowledge and the communication between domain experts and software engineers. The application of such methodologies is known to enhance efficiency and quality of software engineering [1, 2, 3].

Domain-specific modelling tools are meant to support DSM and MDSD by employing model-driven methods in the development (metamodelling) and application (modelling) of DSLs. The Software Technology Group at TU Dresden¹ participated in several research projects addressing different concerns of research in domain-specific modelling tools. During this work, several (meta-) modelling tools were developed. Over time, we found more and more relationships between these tools and possibilities to combine them. Recently, we decided to bundle these tools in a joint software modelling toolbox—the DropsBox. The tools of DropsBox are built on the Eclipse Platform [4] and the Eclipse Modeling Framework (EMF) [5] as a common core. In particular, all tools use the EMF's Ecore as a common metamodelling language which allows a transparent exchange of (meta-) models between different tools. Ecore is aligned with the OMG's EMOF standard [6] for metamodelling.

The DropsBox (like other toolboxes) contributes several diverse and versatile tools for language development and application. Although such a language toolbox already integrates several, individually usable, tools into reasonable collections, it still offers an overwhelming amount of capabilities for development and application of DSLs. There are publications and tutorials for individual tools, however, those are often highly abstract and decoupled from a realistic language engineering approach. From our users we learned that a complete and integrated overview on how to use and combine the tools of such tool boxes is missing. With this paper, we address this problem and provide an overview of DropsBox covering both language design and application. Furthermore, we generalise and relate our experiences to be transferable and interesting for users and developers of other tools and toolboxes.

¹ <http://st.inf.tu-dresden.de/>

The contributions of this paper are as follows. First, we generalise the problems in designing, building, and maintaining DSLs by introducing a DSL life cycle covering both the development and application of DSLs. We discuss activities and relations during individual phases in a DSL's life. This enables a categorisation of tools in the DropsBox and provides readers with a quick overview of their capabilities and application context. Second, we instantiate this life cycle for a continuous example using tools of DropsBox. We show how our tools map to the phases and activities in the DSL life cycle. Thereby, we address the problem of integrated tool application and give readers an impression, how a toolbox can be instantiated for a concrete DSL. Furthermore, we give pointers to tools that can be alternatively used. Third, we address the problem of tool interoperability by discussing the importance of a common integration platform (in our case EMF) to enable a combination of the capabilities of different tools in a toolbox. Fourth, we apply each DropsBox tool to a common example. This covers tools used in DSL development and tools used in DSL application. Readers are provided with material to understand the background for each tool, its application, and its interaction and relation with other DropsBox tools. Fifth, we reflect the challenges and benefits of building and using a toolbox. We report on lessons learned with respect to joint development of tools and their impact on our research, practice of software development, and the education of computer science students.

The remainder of this paper is structured as follows. Sect. 2 introduces the general DSL life cycle. Sect. 3 demonstrates the instantiation of this life cycle with tools from the DropsBox using a common example. Sect. 4 discusses the importance and generalises the characteristics of a common integration toolbox. Afterwards, in Sects. 5–14, we introduce and discuss each tool in turn (cf. Table 1). In Sect. 16, we discuss lessons learned in developing the tools of DropsBox as part of our research activities. The paper is concluded in Sect. 17.

2 DSL Development and Application

Traditionally, approaches to build DSLs originated from artefacts like abstract syntax, concrete syntax and semantics that are involved in language design and implementation. They did not use a particular process or methodology, but solely focused on activities that were required to provide these artefacts. The lack of a systematic approach towards language engineering was identified and addressed by various authors [47, 2, 48]. Mernik et al. stress the importance of an explicit analysis and design phase preceding the implementation of DSLs [2]. When the idea of developing a DSL arises, the DSL authors almost passed the decision phase according to the five development stages from [2]: *decision*, *analysis*, *design*, *implementation* and *deployment*. These phases basically correspond to those described by Antkiewicz et

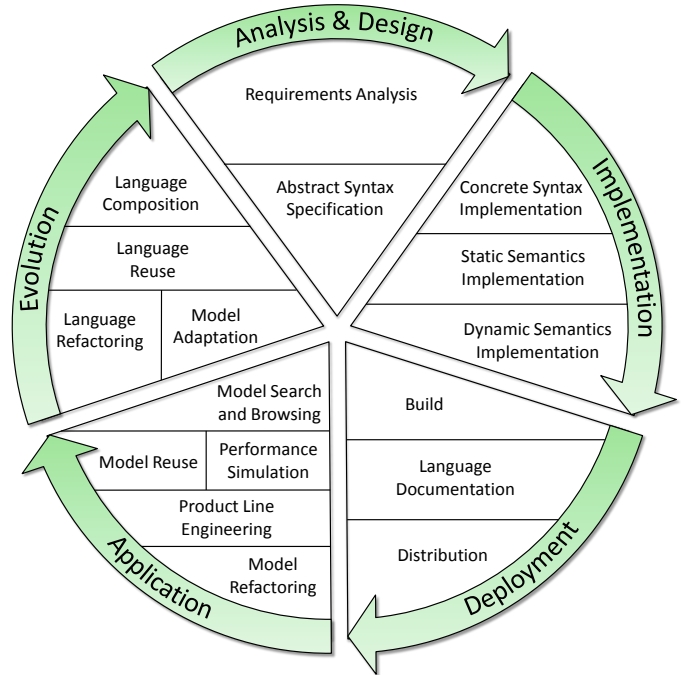


Figure 1 Life cycle for DSL development and application.

al. [48], namely *inception*, *elaboration*, *construction*, and *transition* where inception can be mapped to decision and analysis, and the others to design, construction, and deployment, respectively. Antkiewicz et al. also emphasise the iterative character of DSL engineering.

These processes already cover the main aspects of language building. Additionally, we suggest to consider the application and the evolution of DSLs as vital phases of language engineering that significantly shape their life cycle. To reflect their significance and to show their impact, we propose a refined and extended life cycle for DSL development and application depicted in Fig. 1.

This life cycle starts with the *analysis* of requirements and an initial *design* for the new DSL. Requirements analysis typically covers aspects like language syntax (abstract and concrete), semantics or functionality required with respect to language tooling. In addition, language design requires an in depth specification of the required language concepts and expressions. These are typically declared in the DSL's abstract syntax specification that forms the foundation for all following phases of the DSL life cycle.

During the *implementation* phase, the abstract syntax might be refined and augmented with more implementation detail. Furthermore, this phase is concerned with the implementation of the concrete syntax (i.e., textual or graphical) for the language and language semantics, as earlier defined in the language requirements. This phase is often based on formal specifications ensuring the syntactic and semantic soundness of the DSL. The semantics specification typically consists of two main ingredients: the definition of static semantics including

Tool Name	Capabilities for DSL Life Cycle	Related Tools
EMFText	Generative framework for concrete syntax implementation. Supports <u>language implementation</u> by the generation of parsers, printers, a sophisticated editor, and additional language tooling.	Xtext [7], TCS [8], Monticore [9], Sintaks [10], TEF [11], Spoofax [12]
JastEMF	Adaptation of JastAdd attribute grammars for an application with EMF. Supports <u>language implementation</u> by providing generative tooling to derive an <u>implementation of language semantics</u> from an representation based on attribute grammars.	Silver [13], Kiama [14], FUJABA [15,16], SmartEMF [17]
Dresden OCL	Parsing, interpretation and compilation of OCL constraints for models of several technical spaces. Supports <u>language implementation</u> by enabling the specification and evaluation of well-formedness constraints for metamodels. Supports <u>language evolution</u> by providing a reuseable constraint language that can be integrated in custom DSLs.	Eclipse (MDT) OCL [18], USE [19], SQUAM [20].
DEFT	Documentation of arbitrary models and metamodels. Supports <u>language deployment</u> and <u>language application</u> by providing sophisticated means for creating and maintaining language and artefact documentation.	GenDoc [21], Intent [22]
MDPE Workbench	Model-driven performance analysis. Supports <u>language application</u> by providing means to annotate arbitrary process models with performance data and to apply automated performance analysis.	PUMA [23,24], SPMDA [25].
Reuseware	Introduction of components in arbitrary EMF-based DSLs. Supports <u>language application</u> by providing advanced means in building systems using invasive software composition.	Kompose [26], GeKo [27], GenAWeave [28], MATA [29]
Refactory	Specification and implementation of model refactorings. Supports <u>language application</u> and <u>language evolution</u> by providing generic and customisable refactorings that can be applied for various modelling and metamodeling languages, respectively.	EWL [30], EMF Refactor [31], Operation Recorder [32], GenericMT & Kermeta [33]
Feature-Mapper	Realisation of software product lines using arbitrary DSLs. Supports <u>language application</u> and <u>language evolution</u> by providing variability management for EMF-based modelling and metamodeling languages, respectively.	Model Templates [34], CIDE [35], VML* [36]
Picus	Faceted classification and browsing of arbitrary DSLs. Supports <u>language application</u> and <u>language implementation</u> by providing means for organising and accessing libraries of reuseable DSL and implementation fragments.	MoDisco [37] (Model Browser Component)
LanGems	Role-based language composition system. Supports <u>language design</u> and <u>language evolution</u> by enhancing reuse through language modularity and flexible means for composition.	EMF [5], KM3 [38], Kermeta [39], NetbeansMDR [40], MOFLON [41], MPS [42], MetaEdit+ [43], OSLO [44]. HIVE [45]
JaMoPP	EMFText-based model parser and printer of the Java language. Supports <u>language design</u> and <u>language evolution</u> by allowing the reuse and integration of Java expressions in DSLs.	MoDisco [37], Spoon [46]

Table 1 DropsBox tools, their capabilities and mapping to the DSL life cycle, and related tools with similar capabilities.

well-formedness rules for language instances and the specification of the DSL execution semantics, i.e., its dynamic behaviour. In practise, formal language specifications are frequently replaced by pragmatic approaches like manually developed compilers or interpreters that implement the language semantics.

Once the DSL is implemented, it is typically deployed. The *deployment* phase is concerned with building a deliverable product from the language implementation and distributing it to language users. Another vital aspect of such distribution is to provide appropriate documentation for users to understand the syntax and semantics of the DSL. Therefore, concepts of the DSL itself should be

documented. Hence, tutorials or code examples should be created that describe the practical application of the newly created language.

The most lively and varied phase of the DSL life cycle is its *application*. It involves the creation of language instances to implement concrete software systems and solve certain problems. This phase results in a number of artefacts written in the DSL and has to face a plethora of very specific practical challenges, e.g., refactoring of, component orientation for, performance analysis of, browsing of large sets of, or the creation of product-lines of DSL instances.

The active application typically exposes weaknesses, design flaws and missing features that motivate the *evolution* of the initial DSL design. This was also observed in [49,50]. Such evolution may affect the results of all previously described phases and may trigger a new iteration in the language life cycle. As existing DSL specifications are refined and adapted, language instances need to co-evolve accordingly to conform to the changed specifications. Evolution may also become necessary, when an existing DSL needs to be embedded into a new language. As both, language refinement and embedding, involves the adaptation and integration of languages and language implementations, aspects like language reuse and composition are important for the evolution phase.

Every phase in such a life cycle should be supported by certain tools to support developers and to automate repetitive and error-prone activities. The concrete activities named in Fig. 1 are some examples of a plethora of potential activities. In the next section they will be used to map the functionality provided by the DropBox tools introduced in this paper to the respective development phase they support. The selected activities are not mandatory in all concrete language life cycles and are not considered an exhaustive collection. There is also a large set of related tools that represent alternatives to concrete DropBox tools. Table 1 introduces the DropBox tools covered in this paper, discusses their capabilities for concrete activities of our life cycle they address, and mentions related or alternative tools. For the discussion of related tools we refer to the individual tool sections.

3 Life Cycle Instantiation for an Example DSL

In the previous section, we introduced the general phases and activities of the DSL life cycle. The concrete realisation of its phases and activities heavily depend on the actual DSL and the set of tools employed for its creation. Hence, we consider a selection of tools for developing and applying a concrete DSL as an *instantiation* of the general DSL life cycle. In this section, we illustrate such an instantiation by applying the DropBox tools to the AppFlow, which allows developers to specify the Graphical User Interface (GUI) and user interactions of dialogue-based applications. The AppFlow is used as a common, running example throughout this paper.

3.1 The AppFlow DSL

AppFlow is based on the *Flow Language* used in the open-source framework *flowR*² and inspired by the *Dialog Flow Notation* presented in [51]. A central objective of the DSL is the specification of the dialogue flow, which is defined by a state machine. State machines offer an event-driven, dynamic model for behaviour specification. It allows a separation of user interface components and functionality. The latter can be realised by functions which are called during application execution [52].

The metamodel of the AppFlow language is shown in Fig. 2. Its elements are described in the following. AppFlow **Applications** entail a **StateModel**, which consists of **States**, **Events**, and **Transitions**. Those elements are defined in the **statemodel** package. An **Action** is performed in a **State**. AppFlow defines two types of **Actions** in the **actions** package. One type is used to show **Screens** (**ShowScreenAction**) and the other one is used to call Java methods (**JavaAction**). A **Transition** changes an application's state. It is triggered by an **Event**. An **Event** is sent by a **Button** as the result of a mouse click, or is fired as result of executing a **JavaAction**. The **screenmodel** package contains the **Screen** element. A **Screen** is defined as a **Composite** that contains several **Widgets**. Those dialogue elements are defined in the **widgets** package. Available **Widgets** are **TextFields**, **Texts**, **UILists**, **Buttons**, or **Panels** for nesting **Widgets**. To keep the example comprehensible, data-bindings and the UI layout are not considered in this paper.

3.2 A Project Dashboard Example

To give an impression on how to use the AppFlow DSL, we will shortly present its application to define the dialogue flow of a simple project dashboard, taken from the project management domain. The application lists projects and shows their details after an authorised user login. The main part of the application is a dashboard enumerating a list of current projects and their properties. Users can select a project to get project-specific information and may invoke a help screen.

The textual specification of the project dashboard is shown in Fig. 3. The first line specifies the name of the application—in that case it is simply **App**. The state-model is specified from line 3 to 28. Lines 4 to 9 declare the existing states. The **initial** state marks the entry point of our application. The next line declares the **login** state, which is combined with an action to show the **login** screen. In line 6, **checklogin** is specified to call the static Java method **checkLogin** of the **LoginService** class. Lines 7 to 9 declare the remaining three states **idle**, **help** and **showdetails**. A final state is not directly modelled, since it is implicitly reachable from every state by closing the application.

² <http://www.flowr.org/>

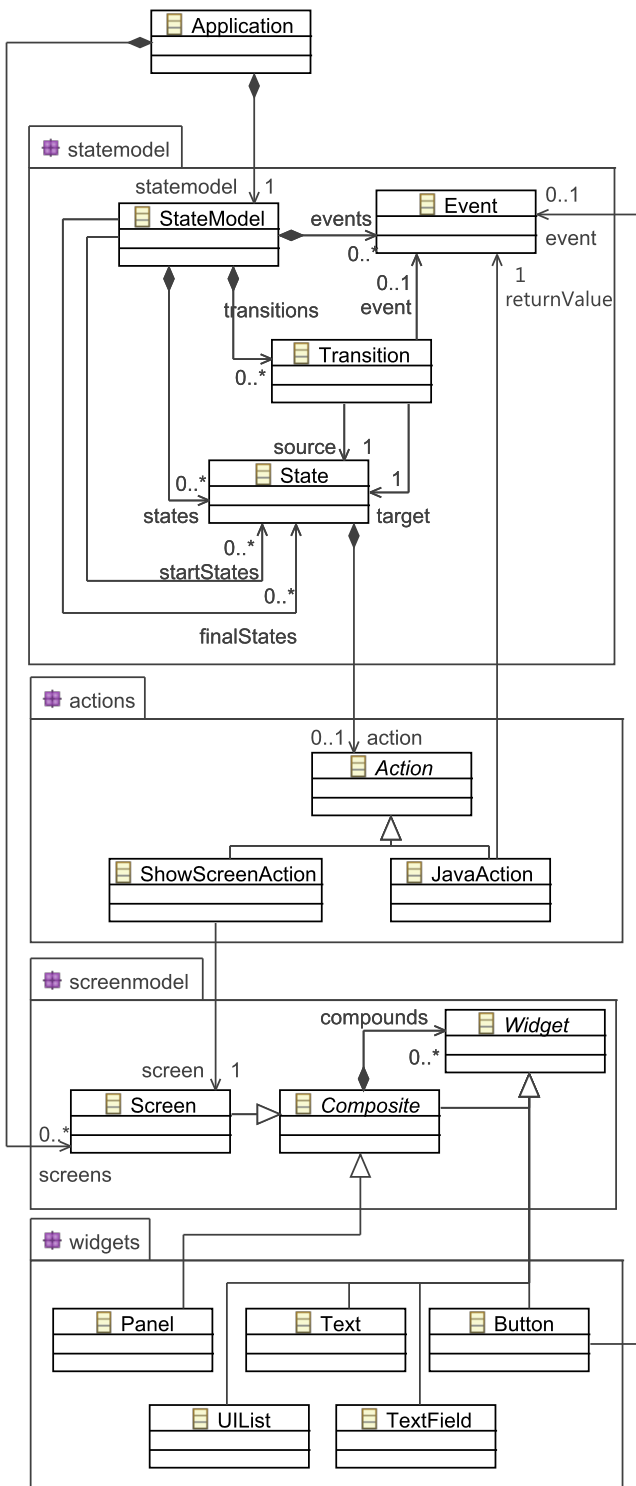


Figure 2 Metamodel of the AppFlow DSL.

Lines 11 to 16 declare events causing transitions, whereas lines 18 to 27 define the actual transitions. A transition statement refers to a source state on the left and target state on the right hand side, separated by \rightarrow . Line 18 shows the transition from **initial** to **login**, which is always executed since no event has been specified. Triggering events can be declared before the transi-

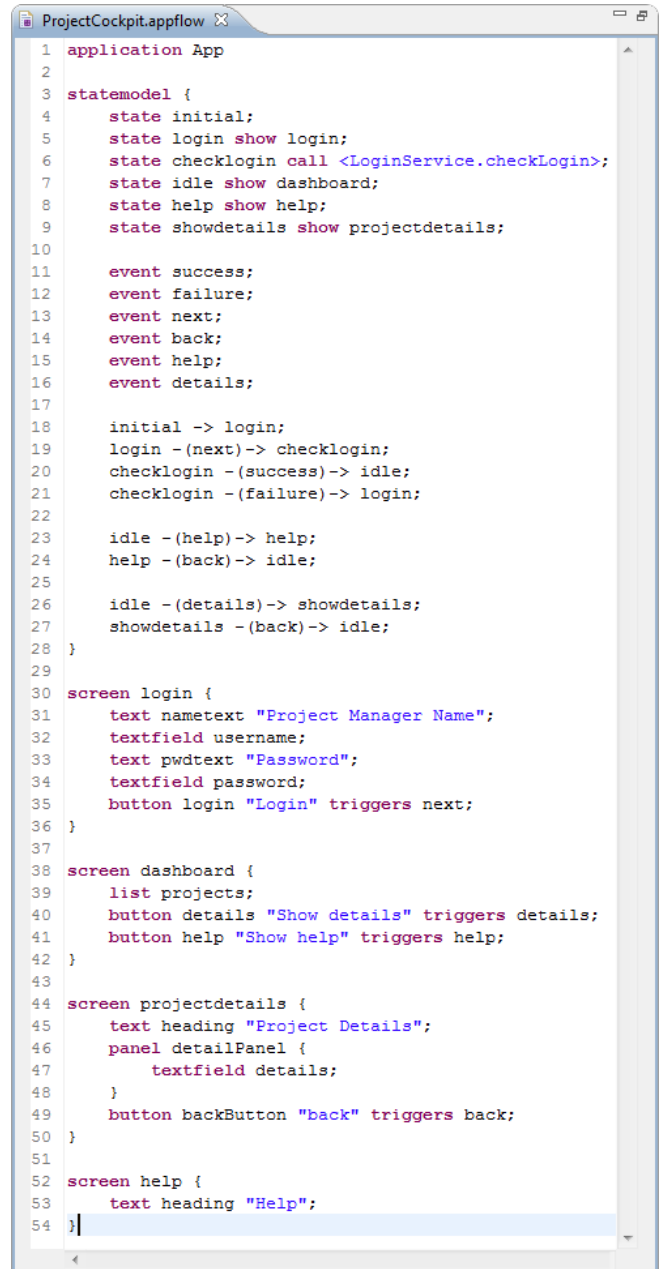


Figure 3 Example dashboard application in AppFlow.

tion symbol, e.g., in line 19 the event **next** is declared as a trigger from **login** to **checklogin**. As shown in lines 20 and 21, a state may have multiple outgoing transitions, all caused by a different event.

The statemodel description is followed by screen definitions. Lines 30 to 36 specify that the **login** screen consists of a text element **nametext** with the label **Project Manager Name**, a textfield **username** for the user's login-name, another label **pwdtext** and a corresponding textfield **password** for the user's password. The **login** button is defined in line 35. If a user pushes this button, **next** is triggered and the state transition defined in line 19 takes effect and leads to the state **checkLogin**.

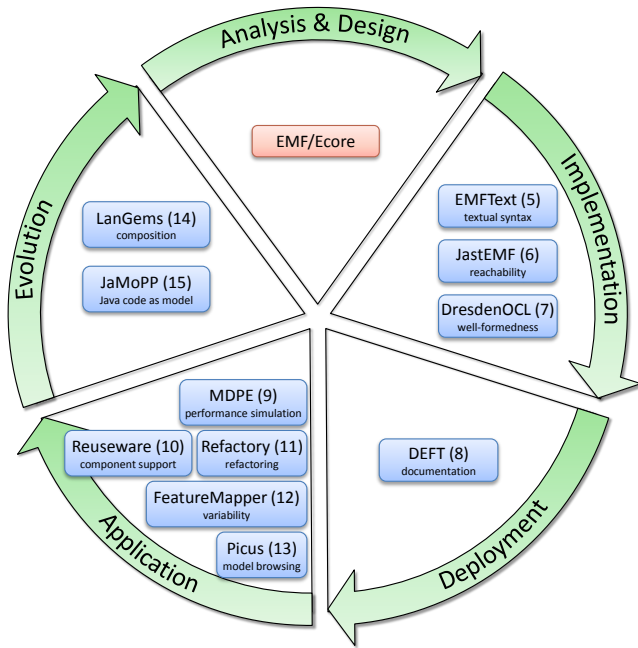


Figure 4 DropsBox tools mapped to the life cycle of the AppFlow DSL. The numbers in brackets refer to the tool’s explanation section in this article.

3.3 Mapping DropsBox to the AppFlow Life Cycle

The instantiation of DropsBox in the life cycle is driven by the needs of the AppFlow language. As indicated in Table 1, the tools are not necessarily restricted to a concrete life cycle phase and could be substituted by equivalent tools or complemented by others. Fig. 4 visualises the mapping of the tools to the concrete life cycle phases of the AppFlow language. In the following, we shortly outline how our tools are used in the development and application AppFlow. Their role within the DropsBox, the concrete application and a comparison to related tools will be discussed afterwards individually in each tool section.

Based on the AppFlow requirements, we specify the abstract syntax in EMF/Ecore in the *analysis and design* phase. As we discuss later in Sect. 4, EMF/Ecore additionally contributes unique technological characteristics that are vital for the implementation of and the interoperability of all tools in the DropsBox. As the requirement analysis for the DSL in this case is an activity without concrete tool support, it is not discussed here.

In the *implementation* phase, a textual syntax for AppFlow is defined and a textual editor for Eclipse is generated using EMFText. In addition, semantics needs to be specified to derive context-sensitive properties and ensure wellformedness of AppFlow instances. In our example, we use JastEMF to implement an attribute grammar for computing the reachable states in a dialogue. On the other hand, Dresden OCL is used to define well-formedness rules on the models.

While there is an overlap in the capabilities of some of the DropsBox tools, we experienced that for some tasks, one tool might be more appropriate for solving the problem than the other. A detailed discussion on how JastEMF and Dresden OCL intersect and on which tasks they can be applied is given in Sect. 6.3.

In the *deployment* phase, we suggest the application of DEFT to create and maintain rich documentation for languages and language expressions to support the users of AppFlow. Other aspects of language deployment like building and distribution are not covered by any DropsBox tool and, therefore, not discussed here.

For the *application* phase, we discuss the employment of DropsBox to address specific practical challenges of AppFlow application. MDPE is used for performance analyses on AppFlow instances. Furthermore, Reuseware adds component support to AppFlow models, whereas Refactory enables refactoring support on the DSL. In addition, Picus is used for browsing a collection of AppFlow model templates, and the FeatureMapper enables product-line engineering with AppFlow.

In the *evolution* phase, tools that enable language evolution are applied. LanGems fosters the modularisation and composition of different DSLs. It is used to derive a modular and more extensible version of AppFlow. JaMoPP lifts the Java programming language to the modelling level and is applied to develop an extension of AppFlow that reuses Java expressions.

4 Architectural Principles of DropsBox

In the previous section we introduced EMF/Ecore as means for syntax specification during language design. In addition, EMF provides the architectural and technological foundation for interoperability among all tools in the DropsBox. All DropsBox tools mapped to life cycle phases can be freely combined without further adaptation steps. Artefacts produced in one tool can be shared with other tools and integrated with artefacts they produce. As tool integration is a significant problem in industry we want to enable a transfer of our experiences to other tool integration scenarios. Therefore, this section analyses which properties of EMF induce interoperability and how they are employed by the DropsBox tools.

Expressive data description language EMF is centered around the expressive data description language Ecore. Ecore allows the specification of graph-structured data with distinct spanning trees. Graphs are feasible to represent data from arbitrary domains and are thus common means to build data structures. As arbitrary graph structures would lead to more complex algorithms (and thus to inefficient tool implementations), each model in Ecore has a unique spanning tree, specified by its meta-models containment hierarchy. This leads to less complex algorithms, efficient traversal strategies and more reliable tool implementations. Relying on a formalism that

is versatile enough to cover a large set of applications, is a prerequisite for successful integration. Basically, all DropsBox tools use Ecore and operate on the induced graph structures.

Strong facilities for reflection EMF has strong facilities for reflection. It allows developers inspecting model instances easily and can therefore enable tools to work with DSLs that are not known beforehand. Reflection is an essential prerequisite to build tools in a more generic way. Tools can adjust their behavior according to the model they shall operate on.

Resource framework EMF provides a resource framework that implements an abstraction layer over concrete data representations. Models can be stored in files, databases or on the network. Tools uniformly access data via Uniform Resource Identifiers (URIs). This abstraction allows the integration of tools that use different means to store data physically, but that operate conceptually on the same domain concepts. Furthermore, the resource framework can be used to bridge technical spaces by, e.g., adaptation as described in [53]. An example for bridging context-free grammars and object-oriented modelling is EMFText (cf. Sect. 5). EMFText extends the EMF resource framework and therefore allows all other DropsBox tools to be applied on textual artefacts. The resource framework does go even one step further; it allows referencing elements in models similar to models as a whole. Thus, tools can reference model elements without knowing about the concrete structure of a model. This fine-grained mechanism for addressing model elements in a uniform way is used by many tools in the DropsBox tool suite. For example, the FeatureMapper tool (cf. Sect. 12) makes heavy use of this property to connect features with model elements.

Limitations of EMF Besides the properties provided by EMF, there are some incomplete or missing features that are important in the context of tool integration.

First, EMF is not entirely programming language independent. Although the concepts of Ecore and its implementation could be transferred to other languages (e.g., C#), the support for custom data types (i.e., Java classes) in Ecore models is a problem when integrating tools written in different programming languages. Since all the DropsBox tools reside in the Java universe, we did not face this problem directly, but it seems obvious that industrial integration scenarios will do so.

Second, the integration of metamodels in EMF is restricted to inheritance and delegation. Since the amalgamation of tool metamodels is often required to establish tool interoperability, more flexible means to combine metamodels are required. For example, a mechanism to directly extend existing metaclasses (e.g., mixins, roles or traits) are certainly required to connect existing tool metamodels.

In summary, we consider the selection of EMF as tool integration platform a good decision. Other toolboxes (e.g., Epsilon [54]) are based on the same foundation and further confirm the feasibility of EMF to build interconnected tools. We think that the properties mentioned above need to be provided by any tool integration platform. In the following sections we will discuss how the tools introduced in our toolbox are combined to implement the AppFlow language. We will discuss the relation of each tool to EMF and highlight interesting relations between DropsBox tools enabled by the integration platform.

5 EMFText

EMFText³ [55] is a tool that can be used to implement textual concrete syntax for modelling languages and to generate Eclipse-integrated textual editors for such languages.

5.1 Background

The initial motivation for the development of EMFText was to ease the creation of instances of Ecore models. Existing approaches were the tree editors generated by EMF and graphical editors built with the Graphical Editing Framework (GEF) or the Graphical Modeling Framework (GMF). The former were hard to use, because editing models was slow and cumbersome. The latter required a lot of effort to be built and were very fragile when changes were made to metamodels.

To build textual languages, many tools (e.g., parser generators) were available, but none of them was tightly integrated with EMF. This was our motivation to design and implement a syntax specification language that was both integrated with Ecore and usable to generate parsers.

Today, EMFText enjoys a wide application in our and others' research and in practise. More than 100 languages are freely available in EMFText's Syntax Zoo [56]. Most prominently, the syntax for Java (cf. Sect. 15), OCL (cf. Sect. 7), and some Ontology-related languages have been developed with EMFText.

Based on an Ecore metamodel, a syntax specification can be derived and customised to generate complex tooling for textual languages. This tooling includes a parser to read models from text files, a printer to serialise models in text format and an Eclipse editor that provides advanced functionality (e.g., syntax highlighting, hyperlinks, error markers, quick fixes, or an outline view). To reuse existing technology, the ANOther Tool for Language Recognition (ANTLR) [57] parser generator was employed to transform the grammar structure to actual Java code.

³ <http://www.emftext.org/>

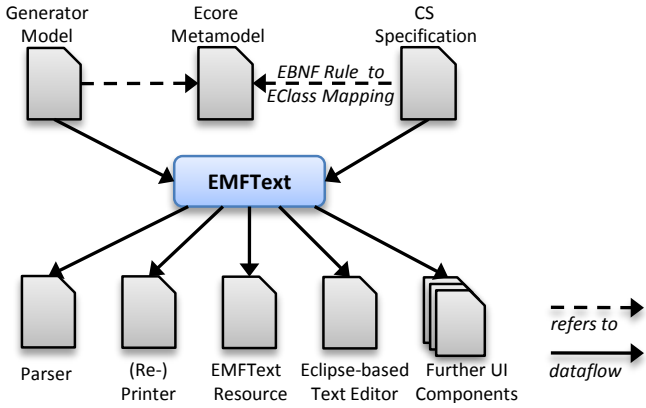


Figure 5 EMFText's tool generation process.

Figure 5 shows EMFText's tool generation process. To create a textual DSL, EMFText requires two central artefacts—the Ecore metamodel (including a generator model) and the syntax specification. The latter is defined using EMFText's syntax specification language CS and is stored in a file with the extension `.cs`.

In its heart, a CS specification contains a context-free grammar (CFG) in a notation similar to Extended Backus Naur Form (EBNF) [58]. Each production in that CFG specifies the textual representation for exactly one metaclass. Furthermore, each CS specification contains token definitions representing the DSL's terminal symbols that are mainly used to specify the syntax of metaclass attributes. Other parts in CS specifications allow to customise code generation and syntax highlighting.

An initial CS specification conforming to the Human Usable Text Notation (HUTN) [59] standard can be automatically derived from the metamodel. This specification can then be refined according to the needs of the language designer. Once both a metamodel and a syntax specification are available, EMFText can generate a set of Eclipse plug-ins that implement the respective tooling for the language. When the syntax or the metamodel changes, these plug-ins must be regenerated.

The generated plug-ins include a parser and a lexical analyser, both derived from the CFG as well as an advanced printer component which allows the serialisation of models according to the CFG in a layout preserving manner. Furthermore, a resource component seamlessly integrates the generated components with the EMF.

Besides the actual editor component, EMFText generates several UI components, including preference pages and wizards.

5.2 Applied on Example

For the running example, we used EMFText to define textual syntax for the AppFlow language. Based on the Ecore metamodel of the language, we defined syntax rules for each concrete metaclass using the CS language.

```

1 SYNTAXDEF appflow
2 FOR <http://www.emftext.org/language/appflow>
3 START Application
4
5 IMPORTS {
6   screen : <http://www.emftext.org/language/appflow/screenmodel>
7   widget : <http://www.emftext.org/language/appflow/widgets>
8   ...
9 }
10 TOKENS {
11   DEFINE COMMENT $'/' (~('\'|\'r'))*$;
12 }
13 TOKENSTYLES {
14   "COMMENT" COLOR #00A000;
15 }
16 RULES {
17   Application ::= "application" name[] statemodel screens*;
18   ...
19   screen.Screen ::= "screen" name[] "{" compounds* "}";
20   widget.Panel ::= "panel" name[] "{" compounds* "}";
21   widget.Button ::= "button" name[] label["", ""];
22   widget.UIList ::= "list" name[] " ";
23   ...
24 }

```

Listing 1 Excerpt of the syntax definition.

An excerpt from this syntax specification is shown in Listing 1.

Here, one can see that the RULES part of the CS language is similar to EBNF, but was slightly adapted to be more usable in the context of syntax definition for meta-models. The RULES language supports the specification of keywords (e.g., “button”), as well as terminals (e.g., `name[]`) and non-terminals (e.g., `compounds`) that refer to structural features of the metaclass. It does also support repetition operators (e.g., asterisk, question mark and plus).

The TOKENS part contains exactly one token definition for COMMENTS, while all other token definitions are added by default (e.g., EMFText includes identifiers by default) or are derived from the specification automatically. In the TOKENSTYLES section, the default colour for COMMENTS is specified. Note that at runtime this can be changed in the EMFText editor preference page.

In the IMPORTS section different namespaces of the AppFlow language are declared such that metaclasses within these packages can be uniquely referenced. For example, `Application` belongs to the default namespace while `Screen` belongs to the `screen` package and the remaining rules belong to the `widget` package.

Presenting all the details about the CS language is out of the scope of this paper, but they can be found in the EMFText User Guide, which is available from the EMFText homepage.

After defining the syntax for the AppFlow language, we generated tooling to conveniently edit and process AppFlow models as shown in Fig. 6.

The parser that is generated by EMFText is automatically registered with the EMF resource framework and thus used by all EMF tools to read textual AppFlow models. Therefore, arbitrary EMF-based tools—not only the ones presented in this paper—can handle AppFlow models. Besides the serialisation functionality and the

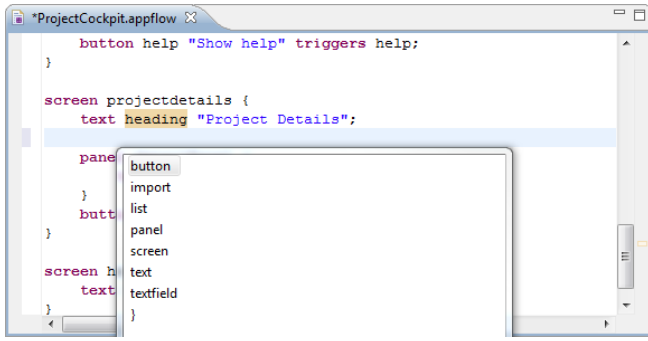


Figure 6 EMFText generated editor with code completion.

generated editor, EMFText does also provide additional infrastructure code that is commonly required for DSLs. For example, a so-called builder is generated that is invoked whenever AppFlow models change. Builders are frequently used to compile models (i.e., to automatically derive other artefacts from the modified model). Also, extension points are generated that allow registering post processors. These can be used to validate models automatically after loading them.

5.3 Role in DropsBox

EMFText plays a central role in DropsBox. First, it allows us defining new textual modelling languages quickly (e.g., to build example models that can be processed by other tools). Moreover, EMFText is heavily used to build other DropsBox tools. Refactory (cf. Sect. 11) relies on three modelling languages that were built with EMFText. Dresden OCL (cf. Sect. 7) provides an advanced OCL editor that was generated by EMFText. JaMoPP (cf. Sect. 15) relies on code generated by EMFText to create models from Java source code. Also, LanGems (cf. Sect. 14) and Reuseware (cf. Sect. 10) employ textual modelling languages built with EMFText. Besides EMF itself, EMFText is one of the foundations that almost all DropsBox tools rely on.

5.4 Related Tools

A variety of different tools to specify textual concrete syntax exists. Xtext [7] and TCS [8] are parser generators for EMF that are quite similar to EMFText. Monticore [9] provides a metamodeling language that supports an integrated specification of concrete and abstract syntax. Sintaks [10] is a parser generator for the Kermeta metamodeling language. TEF [11] provides an interpretative approach to concrete syntax implementation using a Model View Controller (MVC) update strategy. Most of these tools focus on syntax implementation and are not contained in a toolbox for supporting the DSL life

cycle in general. Xtext is an exception, as it provides further languages to check well-formedness rules for meta-models, for model transformation and code generation. For a more detailed comparison we refer to [55] and [60].

6 JastEMF

JastEMF⁴ [61,62] is a tool that permits the specification of static semantics of Ecore metamodels using reference attribute grammars (RAGs). It is used in the implementation phase of the DSL life cycle.

6.1 Background

Attribute grammars (AGs) [63,64] are a well-known formalism to specify the static semantics of context-free languages. Hence, static program analysis like name and type analyses are typical applications of AGs. An AG associates each non-terminal of a given CFG with a set of *synthesised* and *inherited* attributes, which represent its semantics. Given an AG G_{AG} for a CFG G_{CFG} an attribute grammar system can be used to generate an attribute evaluator that computes for every possible derivation tree of G_{CFG} —called abstract syntax tree (AST) in the following—the value of all the attributes associated with its non-terminal nodes. Thus, the evaluator computes the semantics of the AST. The semantics are specified in attribute equations. For every production p of G_{CFG} an equation must be given for all the *synthesised* attributes of the left-hand side non-terminal and for every *inherited* attribute of the right-hand side non-terminals. The equation can depend on any attributes of the production. Thus, *synthesised* and *inherited* attributes represent data-flow up- or downwards an AST, respectively. By combining *synthesised* and *inherited* attributes in equations, arbitrary dependencies between tree parts can be specified.

Since their first presentation in 1968, the AG formalism and evaluation strategies have been steadily extended and improved, such that AGs nowadays achieve a good abstraction on the specification level (i.e., convenient specifications) while still supporting efficient evaluators (i.e., high-performance implementations). In general, AGs are a good choice for the implementation of static language semantics of DSLs [65].

One recent attribute grammar system is the JastAdd metacompiler [66], an object-oriented attribute grammar system based on Java that supports advanced AG concepts such as reference, collection and circular attributes. Reference attributes [67] provide means to associate remotely located AST parts with each other, such that attribute equations can depend on non-local attributes. Thus, reference attributes impose a graph on

⁴ <http://www.jastemf.org/>

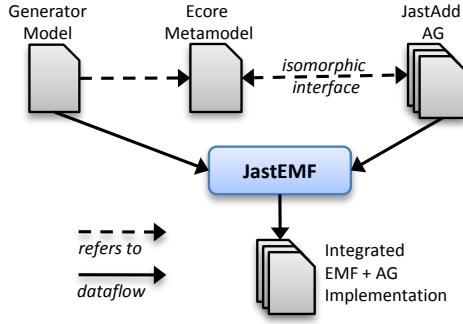


Figure 7 JastEMF code generation process.

top of the AST (the abstract syntax graph (ASG)). Collection attributes [68] permit the efficient specification of attributes, whose values depend on different AST parts. Finally, circular attributes are convenient to specify fixpoint semantics [69]. Especially in combination with reference attributes, fixpoint semantics ease the computation of transitive closures, which is often required for advanced language analysis like dataflow analysis.

Currently, modelling frameworks such as the EMF are heavily used in DSL development. However, while being adequate for defining structure, they lack proper formalisms to specify the computation of semantic values. Our JastEMF tool unifies the attribute grammar and the metamodeling worlds to achieve semantics integrated metamodeling [61]. The basic idea of the approach is distinguishing syntactic and semantic interface of a metamodel according to the attribute grammar formalism. The syntactic interface is given by its metaclasses, inheritance relations, non-derived properties and containment references, which specify context-free structure. On the other hand, metamodels usually contain so called non-containment (i.e., cross) references, derived properties and operations forming a semantic interface. Typically, things like reference resolving based on name analysis, types and closures are declared in this interface.

Figure 7 shows how JastEMF works. Since it is based on the EMF and the JastAdd metacompiler, it requires the Ecore and generator model of the language, which declare the syntactic and semantic interfaces. The actual AG must be given by a set of JastAdd specifications. The context-free structure is specified by a JastAdd AST specification which has to be equivalent to the containment hierarchy defined in the Ecore model. Note that JastEMF can derive the AST specification automatically. Static semantics is specified by a set of JastAdd *jrag* specifications. Each *jrag* specification contains a set of semantic aspects consisting of attribute declarations and equations, where each equation is a Java expression or method body. Given a set of well-formed specifications, JastEMF triggers the EMF and JastAdd code generators and integrates the resulting class hierarchies. The result is a semantics integrated metamodel implementation. Every method stub the EMF generates

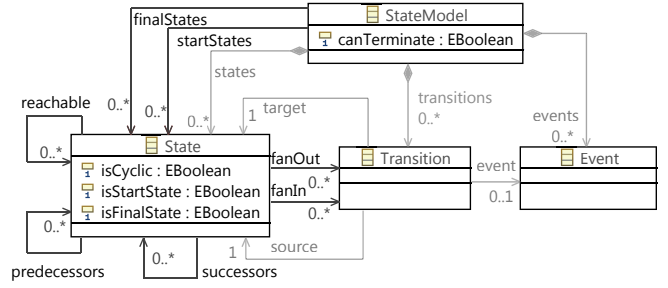


Figure 8 Semantic interface of the statemodel.

for derived properties, non-containment references and operations is replaced by its JastAdd evaluator implementation. For more details on the integration process, we refer to [61].

6.2 Applied on Example

For the running example, we focus on the StateModel sub-language of the AppFlow language. We used a RAG to solve the following problems:

- Specify if a state is an initial or a final state.
- Collect initial and final states in the AST and make them available at the StateModel root.
- Derive incoming and outgoing transitions of states as well as their direct predecessors and successors.
- Compute transitive closures.
- Specify if a state is contained in a cycle.
- Compute if the statemodel can terminate, i.e., if a final state can be reached.

The corresponding parts of the AppFlow metamodel are shown in Figure 8. Parts of the semantic interface are coloured in black while others are greyed out. Listing 2 contains the respective attribute declarations in JastAdd while Listing 3 contains their realisation.

Each **State** inherits its incoming and outgoing transitions (**fanIn** and **fanOut**) from its parent **StateModel**. These attributes are declared being *lazy* which enables caching and avoids redundant computations. A reference to the **StateModel** root is also passed to each **State**, however, it is only a supporting attribute and does not belong to the semantic interface of the Ecore model. In the example, we also use *collection* attributes. Each start and final **State** contributes itself to the **StateModel**'s **startStates** or **finalStates** collection attribute, respectively. The **predecessors** and **successors** relations are directly computed from the **fanIn** and **fanOut** attributes by iterating over the **Transition** lists computed by these attributes and adding the corresponding **States** to the result. **Reachable** represents the transitive closure of the successor relation (i.e., it computes the set of reachable **States** for a given **State**). Since this kind of computation requires fixpoint semantics, **reachable** is a *circular* attribute. Based on it, **canTerminate** computes if a final **State** can be reached from the **startState**.

```

1 aspect Reachability{
2   inh StateModel State.stateModel();
3   coll EList StateModel.startStates()
4     [new BasicEList()] with add;
5   coll EList StateModel.finalStates()
6     [new BasicEList()] with add;
7   syn lazy boolean StateModel.canTerminate();
8
9   inh lazy EList State.fanOut();
10  inh lazy EList State.fanIn();
11  syn lazy EList State.successors();
12  syn lazy EList State.predecessors();
13  syn lazy EList State.reachable()
14    circular [new BasicEList()];
15
16  syn boolean State.isCyclic();
17  syn boolean State.isStartState();
18  syn boolean State.isFinalState();
19 }

```

Listing 2 Semantic interface in JastAdd.

```

1 aspect Reachability{
2   eq State.isCyclic() = reachable().contains(this);
3   eq State.isStartState() = predecessors().size() == 0;
4   eq State.isFinalState() = successors().size() == 0;
5
6   eq State.successors() {...}
7   eq State.predecessors() {...}
8   eq State.reachable() {...}
9
10  eq StateModel.getstates(int index).stateModel() = this;
11  eq StateModel.getstates(int index).fanOut() {...}
12  eq StateModel.getstates(int index).fanIn() {...}
13
14  State contributes this when isStartState()
15    to StateModel.startStates() for stateModel();
16
17  State contributes this when isFinalState()
18    to StateModel.finalStates() for stateModel();
19
20  eq StateModel.canTerminate(){...}
21 }

```

Listing 3 Excerpt from attribute grammar specification.

6.3 Role in DropsBox

JastEMF provides convenient means to specify meta-model semantics. Thus, any EMF tool requiring complex semantic computations profits from JastEMF. In the running example all other tools can reuse the semantics JastEMF generated into the metamodel implementation. Especially tools only concerned about syntax, like EMFText, conveniently cooperate with JastEMF. Since the EMFText parser constructs a model M while parsing, the semantics integrated by JastEMF are available for all the tools that use M .

Similar to AGs, the application area of the OCL is also static language semantics (e.g., reachability could also be specified using OCL definitions). Since JastEMF is based on the JastAdd AG system, its computational expressiveness is similar to that of the OCL. In fact, both formalisms are Turing complete and can be used for simulating a Turing machine by introducing recursive functions using the **def** keyword in OCL and via non-terminal or circular attributes in JastEMF.

However, there are some differences between both approaches with regard to problem abstraction and their view on language semantics. AGs are well suited for specifying computations depending on values distributed over a model instance. Dataflow can easily be specified using inherited or synthesised attributes, while reference attributes are a good choice for specifying derived non-containment references. Complementary, the OCL as a functional constraint language is well suited to declare constraints over model instances in a non-verbose way and for providing appropriate user feedback. However, as our experience with OCL indicates, it is not well suited for computing complex relations from sets of arbitrary non-local model elements (e.g., elements distributed over complex, nested block structures).

In Sect. 7, we will use OCL to check additional properties previously computed by the AG and generate error markers and warnings that are prompted to the user of the textual editor generated by EMFText.

6.4 Related Work

We are not aware of any approach that integrates a metamodeling language like Ecore and AGs. However, there are several tools that provide a similar approach to AGs as realised by JastEMF/JastAdd. Silver [13] is a demand-driven AG system which provides its own functional language for attribute equations and its own parser generator. As JastAdd, it supports collection, reference and reference attributes and provides a module system.

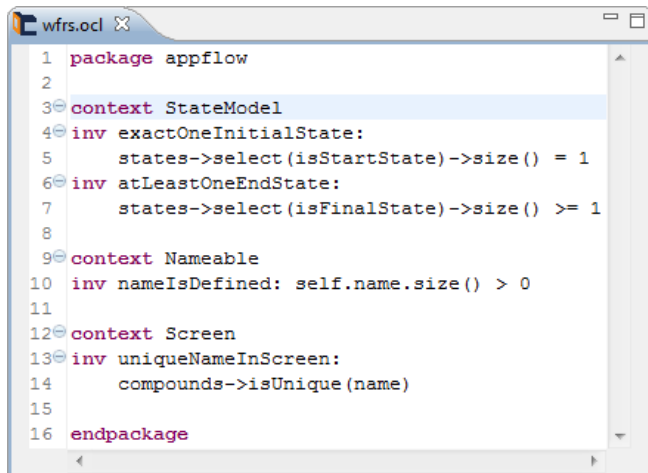
Kiama [14] is a library that embeds AG-based language processing capabilities into the Scala [70] programming language. Essentially, it provides the same AG features as JastAdd. However, in contrast to JastEMF, abstract syntax and attribute equations have to be specified using Scala features. As a consequence, users have to take care of AG well-formedness without tool support.

Beyond AGs, several approaches for specifying meta-model semantics exist. Fujaba [15,16] provides a graphical language for defining rewrite rules on EMF models and thus can be used to specify static semantics of a DSL. In [71], Abstract State Machines (ASMs) have been proposed to specify metamodel semantics using semantic anchoring. However, while ASMs are good for specifying execution semantics, static semantics is not covered by the approach. SmartEMF [17] uses Prolog for constraint checking of operations on XML files that have been mapped to an EMF model instance. SmartEMF also resolves references of weakly coupled DSLs in multiple domains. This is currently not supported by JastEMF.

7 Dresden OCL

Dresden OCL⁵ [72,73,74] allows developers defining and editing OCL constraints and queries on different (meta-) models and evaluating them on various instances.

⁵ <http://www.dresden-ocl.org/>



```

1 package appflow
2
3 context StateModel
4 inv exactOneInitialState:
5   states->select(isStartState)->size() = 1
6 inv atLeastOneEndState:
7   states->select(isFinalState)->size() >= 1
8
9 context Nameable
10 inv nameIsDefined: self.name.size() > 0
11
12 context Screen
13 inv uniqueNameInScreen:
14   compounds->isUnique(name)
15
16 endpackage

```

Figure 9 WFRs specified in OCL.

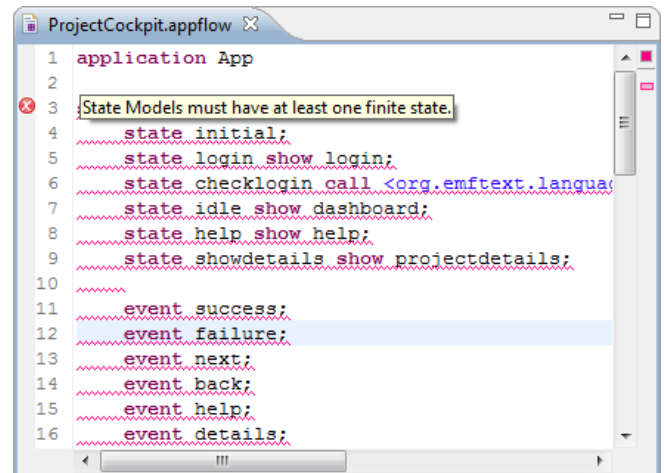
7.1 Background

The Object Constraint Language (OCL) [75] is an OMG standardised language to specify constraints on meta-models and models using concepts from *design by contract* [76]. Today, OCL is a well-established constraint language that is often used in the context of UML as well as of DSLs.

To support the development of OCL and its application in model-driven engineering, we implemented Dresden OCL—a toolkit for parsing and evaluating OCL on models of various technical spaces. Dresden OCL is based on Eclipse/EMF but supports the parsing of OCL constraints defined on EMF and UML models as well as Java and XSD files. OCL can be evaluated by either using the Dresden OCL interpreter for various technical spaces [74] or by using the Dresden OCL AspectJ/Java and SQL compilers [77]. Dresden OCL has been integrated into multiple modelling tools including ArgoUML⁶ and MagicDraw UML⁷. Dresden OCL is already well tested using more than 3500 JUnit test cases as well as multiple case studies [74, 78].

7.2 Applied on Example

In the context of the DSL life cycle introduced in Sect. 2, Dresden OCL can be used in the implementation and application phases. During implementation, well-formedness rules (WFRs) can be defined on the language’s metamodel that are later evaluated for instances of the language. Alternatively, for DSLs being modelling languages themselves, Dresden OCL can be applied during the application phase (e.g., the AppFlow could be extended to define OCL conditions on transitions between different states). However, in the AppFlow example presented here, Dresden OCL is used during the implementation phase to define and evaluate WFRs on top of



```

1 application App
2
3 [State Models must have at least one finite state.]
4 state initial;
5 state login show login;
6 state checklogin call <org.emftext.language
7 state idle show dashboard;
8 state help show help;
9 state showdetails show projectdetails;
10
11 event success;
12 event failure;
13 event next;
14 event back;
15 event help;
16 event details;

```

Figure 10 WFR evaluation on the AppFlow example.

the AppFlow metamodel. For example, each **StateModel** must have exactly one initial **State** and at least one finite **State**. Whether **States** can be considered as initial or finite could be expressed using OCL, but JastEMF already provides derived attributes containing this information (cf. Sect. 6) and therefore, they can be reused within the OCL constraints. Figure 9 shows some OCL statements. Besides the invariants checking for the right quantity of initial and final states (lines 3–7) it is ensured that every **Nameable** element has a non-empty name (lines 9–10) and that such names are unique within a **StateModel** (lines 12–15).

Dresden OCL allows the parsing of OCL constraints defined on Ecore models, thus no further effort is necessary to define constraints on the AppFlow metamodel. The AppFlow metamodel can be imported as a model into Dresden OCL and OCL WFRs can be parsed. The integration of the OCL WFRs within the AppFlow language is rather simple. EMFText provides the possibility to implement postprocessors that can be used to modify or check parsed models. Such a postprocessor has been implemented which uses the parser and interpreter of Dresden OCL to parse the WFRs on the AppFlow metamodel and verify them against parsed AppFlow models. If a WFR is violated, an error message is created and displayed within the AppFlow editor (cf. Fig. 10).

7.3 Role in DropsBox

Within DropsBox, Dresden OCL currently uses the tooling provided by EMFText (cf. Sect. 5), as the OCL Parser/Editor of Dresden OCL was built using EMFText and can be considered as one of the biggest case studies evaluating the capabilities of EMFText. Refactorings have been added to the Dresden OCL Editor (e.g., *rename* or *extract definition*) using Refactory (cf. Sect. 11). Besides, Dresden OCL can reuse attribute values computed by JastEMF (cf. Sect. 6), as shown in the AppFlow example. Furthermore, OCL has been integrated into

⁶ <http://argouml.tigris.org/>

⁷ <http://www.magicdraw.com/>

other tools of DropsBox. Reuseware (cf. Sect. 10) integrates OCL into its specification languages. FeatureMapper (cf. Sect. 12) uses an extended version of OCL to check the well-formedness of members of software product lines.

7.4 Related Tools

Besides Dresden OCL, a large amount of other OCL tools exist. The most popular ones are shortly outlined below. The most widely used OCL tool today is Eclipse (MDT) OCL [18]. It provides parsing and interpretation capabilities for EMF and UML, but no OCL compilers. Another popular OCL tool is USE that supports parsing and interpretation of OCL constraints on a subset of UML [19]. USE is not based on EMF but uses its own metamodel that can be considered as a subset of UML. SQUAM OCL focuses on extending OCL with library and unit testing support [20]. It is built as an extension of other OCL tools, supporting reuse of both Dresden OCL and Eclipse OCL. A list of further OCL tools and a comparison of them can be found in [18].

8 DEFT

DEFT⁸ [79, 80] stands for “Development Environment For Tutorials”. It is a tool for the creation and maintenance of external software documentation. As such it is part of the deployment phase and the application phase of the DSL life cycle.

8.1 Background

External software documentation is documentation which is not directly attached to software fragments, for example programming tutorials, architectural overviews or end-user documentation. It consists to a great degree of running text, which is interspersed with figures, such as code listings or images of model diagrams.

Usually, software continues to evolve after documentation is written. Hence, to keep up with the changes, documentation has to evolve, too. However, keeping external documentation synchronised with the software is tedious and error-prone and should be supported by appropriate tooling. A generative approach that derives the final document(s) from a set of specifications and modelling artefacts can help here. Often, the generation is driven by templates which determine structure and layout of the generated documentation. Hence, if a software artefact is modified, the documentation can be easily regenerated. However, there are some problems with this approach. Often it would be useful to add textual explanations to the generated documentation (e.g., describing

figures, explaining tutorial steps, etc.). Since these explanations are not part of the original templates, they will be lost if the document is regenerated and, thus, have to be added manually again.

To overcome this problem for source code documentation, the paradigm *Elucidative Programming* [81] has been introduced. Documentation is mainly text written manually which is meant to be interspersed with code listings or hyperlinks to source code. These code listings or hyperlinks are not directly included in the documentation, though. A placeholder reference pointing to a specific code fragment (e.g., a method) is added instead. In a subsequent rendering step, the references are replaced by hyperlinks or by code listings, respectively. If the documentation has to be regenerated only the references are reevaluated while the explanatory text is left untouched.

In order to reference specific parts of code, *Elucidative Programming* environments are language-specific. This hampered reuse of previous tools. DEFT extends the Elucidative Programming approach in various ways:

1. Besides plain source code, DEFT works with any kind of structured data (e.g., models, ontologies, formalised requirements). For each type of artefact, a number of rendering transformations generate representations that can be displayed in a document. For example, source code can be represented by code listings with styled text, indentation and syntax highlighting. Models and ontologies can be represented as bitmaps, vector graphics, or as tables listing various attributes. Relationships between requirements in formalised requirement models can be represented as a traceability matrix.
2. DEFT provides an integrated environment for documentation writing and maintenance. There are various integrated editors, such as OpenOffice, the Visual Editor for XML (VEX), or the LaTeX editor Texlipse. The editors display generated artefact representations instead of the plain references. This comprises both WYSIWYG (“What You See Is What You Get”), such as in OpenOffice, and markup, such as in Texlipse. Therefore, the documentation author is not burdened with special syntax of the references, as in normal template generation approaches.
3. DEFT supports hot update of documentation. When documented software artefacts are modified, the documentation is immediately updated. This is achieved by the reevaluation of all affected references and the insertion of the newly generated representations into the document. This lets the documentation writer always see an up-to-date version of the documentation.

⁸ <http://deftproject.org/>

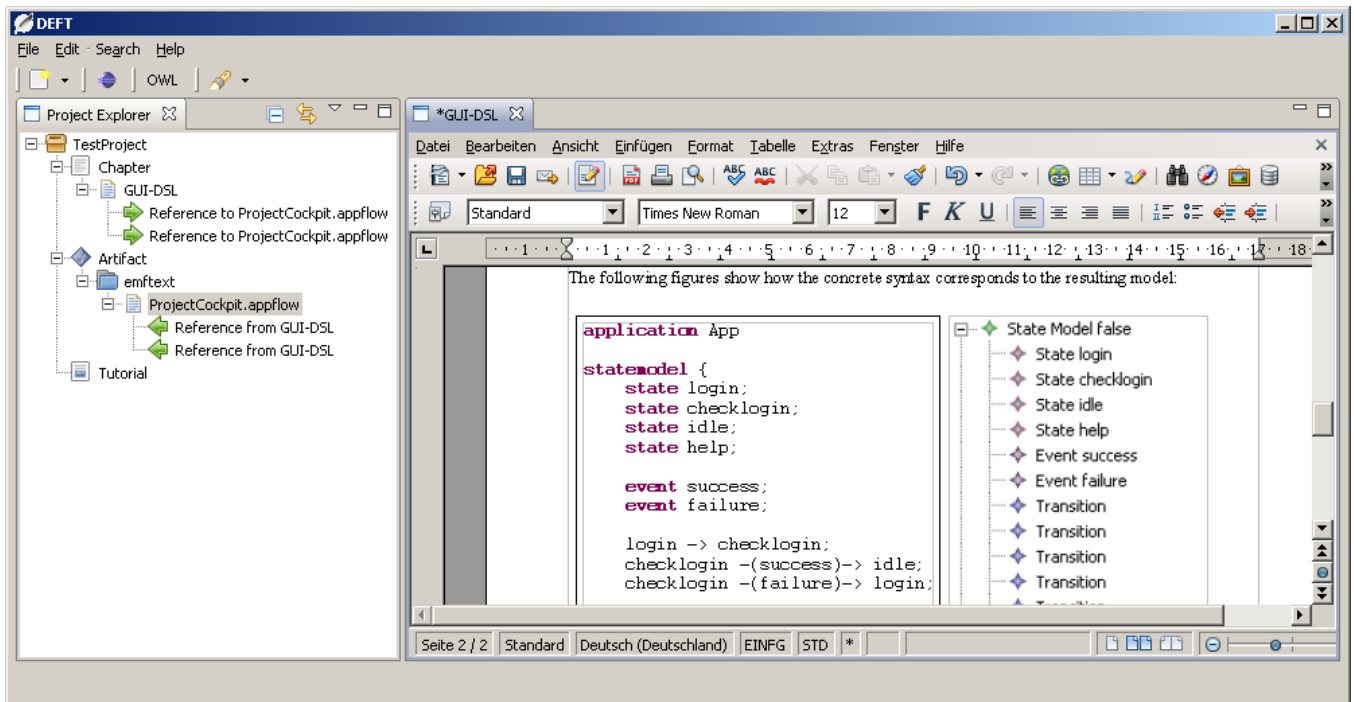


Figure 11 Screenshot of DEFT with embedded text and two generated model representations.

8.2 Applied on Example

For our running example, DEFT has been prepared to deal with the AppFlow language. Since support for EMF-Text is already included, adding a new EMFText-based DSL has been a matter of minutes.

DEFT can be applied in the AppFlow example in two ways. First, a tutorial helps new AppFlow users getting familiar with the language. Explanations which are easy to understand together with examples greatly reduce the training effort and lead to a better confidence in DSL usage. This case belongs to the deployment phase of the DSL life cycle.

Second, system documentation is helpful for future developers of a system because it can provide an overview which parts of the GUI were created from which models, and what the concrete specifications look like. The writing of system documentation belongs to the application phase of the DSL life cycle.

Figure 11 shows a screenshot of DEFT, displaying a code listing and a tree representation of an AppFlow model side by side. Unlike plain text, the code listing and image of the tree representation are not physically embedded in the documentation but referenced and rendered as images. The following steps have been performed to create the documentation.

1. First the AppFlow file has been imported into the DEFT repository.
2. Then a document was created in the repository and documentation text was written into the document.
3. Afterwards two references to the AppFlow file were inserted into the document. The references have been

parameterised with a GUI wizard to control the final appearance of the model in the documentation. For example, in the case of the textual model description it was possible to select only excerpts that should be added to the documentation, or to define the font size. For the model tree it was specified which nodes should be expanded and collapsed.

4. The references were then hidden and the generated representations were displayed immediately, namely styled text for the code listing and an image for the model tree. This relieved the author from dealing with the complex internal format of the references.

The location of the original models is stored in the repository along with the copies of the content. Thus, it is easy to detect when an artefact has been modified or removed. If such a change or removal is found, the documentation writer is informed and can choose whether to update the repository. After updating, all artefact representations will be recomputed from the new versions of the artefacts in the repository. If artefacts have been modified in complex ways, the resulting representations may differ from the writer's intention. If artefacts have been deleted, it is not possible to compute a representation at all. Hence it is necessary to check the documentation for correctness. Therefore, an overview of all artefact references in the documentation which are affected by the update is presented. This way, a manual search for defective parts in the documentation is avoided.

However, the running text is not explicitly considered. Therefore, it is advisable to have figures and explanatory text close to each other. In this case, it is sufficient to check if the text in the proximity of changed

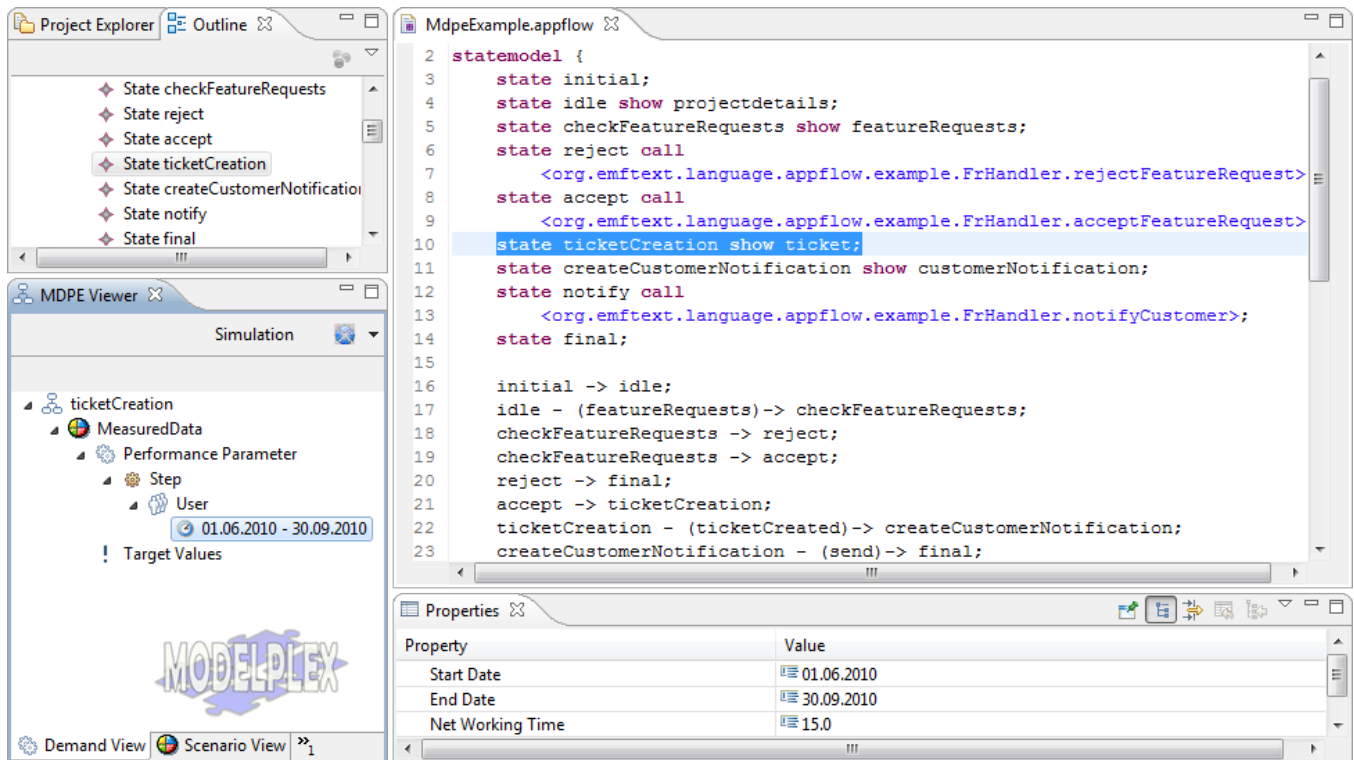


Figure 12 State chart to handle feature requests.

figures or cross references to these figures is still consistent and to update it if necessary.

8.3 Role in DropsBox

DEFT is a tool for the generation and maintenance of documentation which can be partly generated. It can generate representations for a variety of software artefacts, most notably Java source code, Ecore models and EMFText-based DSLs, including the AppFlow language and OCL. DEFT reuses parts from the generated EMFText plug-ins to derive artefact representations. For example, the syntax highlighting component in the DSL editor is reused to render the syntax highlighting for code representation. The **TreeViewer** of the EMFText outline, which is used to display the model's structure, is reused to create image representations.

8.4 Related Tools

GenDoc [21] is a template-based documentation tool for UML diagrams made with the Topcased UML editor. It can generate MS Word and OpenOffice documentation.

Intent [22] is a new Eclipse documentation project which has been inspired by Literate Programming [82]. It enables the documentation of many software artefacts with the help of a special documentation language. The documented artefacts are automatically transformed into EMF models where constraints can be checked.

9 MDPE Workbench

The model-driven performance engineering (MDPE)⁹ workbench [83] is a tool that allows the extension of arbitrary process modelling tools with functionality for performance engineering. It can be used to check performance requirements and to detect performance bottlenecks in process models.

9.1 Background

In the case of applying performance engineering in the process modelling domain, performance models are mostly created manually from process models by using specific performance analysis tools. Such an approach implies additional effort and costs due to manual creation of performance models as well as due to model changes, which either are a result of a performance analysis or a result of an extension of the process model.

The central motivation behind MDPE is to bridge the gap between the process modelling domain and the performance engineering domain. This means that performance analyses are applied directly on process models, which are created by domain experts. Performance models are derived from annotated process models in an automatic manner. These performance models are analysed and the results are returned as performance

⁹ <http://www.mdpe.org/>

feedback directly into the original process development environment.

In MDPE, the process modelling and the performance engineering domains are bridged by providing a so-called tool-independent performance model (TIPM) [84] and applying model transformation and tracing techniques. A TIPM is independent from any modelling language as well as from any performance analysis tool. It is generated from a process model, which is annotated with additional performance data (e.g., execution times, branch probabilities, resource demands). In a second step a so-called tool-specific performance model (TSPM) is generated from the TIPM. The TSPM can be executed or interpreted directly by external analysis tools like AnyLogic [85]. Both transformation steps are implemented as model-to-model transformations by using the ATLAS transformation language (ATL) [86]. Tracing techniques are used to propagate analysis results back to the original process model. Technically, this is realised by using trace models [87] based on a model weaving approach [88].

9.2 Applied on Example

In the context of the project management example, we assume that the AppFlow language is used to develop user interfaces for a business process (e.g., a BPMN process) responsible for managing feature requests of customers. AppFlow applications are associated with manual tasks of the business process. MDPE can help to identify execution times of single process tasks by capturing performance data in AppFlow applications and executing performance analyses. For instance, in case of an increased workload of the process due to a new project, performance analyses can be used to detect performance bottlenecks in participating departments of the process. Thus the MDPE workbench is applied to the AppFlow during the application phase of the DSL life cycle as outlined in Sect. 2.

Figure 12 shows the MDPE workbench. The AppFlow editor shows the state chart of the application for the process task to handle new feature requests from customers. Requests are reviewed by a project manager, who is responsible to create and assign tickets. The manager is able to show (**idle**) and check received feature requests from customers (**checkFeatureRequest**). Each request can be accepted (**accept**) or rejected (**reject**). In case of accepted feature requests, the manager has to create a ticket (**ticketCreation**). Furthermore, the manager must create a notification message for the customer (**createCustomerNotification**), which is sent automatically (**notify**). To analyse the end-to-end execution time for the complete task to handle a feature request, performance data (e.g., execution times) for states with human interaction and transitions need to be annotated to AppFlow model elements. Performance data can, for instance, be estimated, captured through experiments, or collected based on monitoring data.

Technically, the performance data is annotated by using the MDPE Viewer, which is shown in the lower left part of Fig. 12. It shows annotated resource demands for the **ticketCreation** state, which is executed by a resource **User**. The properties view shows that this state has an execution time of 15 minutes for each feature request. It has to be noted that multiple periods with different performance data can be defined to specify data in a fine-grained manner. States with more than one outgoing transition (e.g., the outgoing transitions of **checkFeatureRequests**) require that each transition is annotated with a branch probability. Initial states must be annotated with a workload description.

After appropriate performance data was defined, the analysis can be issued, which means that a TIPM and a TSPM model are generated. The TSPM is simulated and the feedback is presented in the MDPE viewer. In the example, this is the mean end-to-end execution time to handle new feature requests.

9.3 Role in DropsBox

The MDPE Workbench works with EMF-based languages. So far, our tool provides adapters for MARTE-annotated [89] UML activities and a more flexible approach by using annotation models. While MARTE is limited to UML, annotation models can be adapted to arbitrary languages like AppFlow. MDPE can be combined with Reuseware (cf. Sect. 10) for analysing the performance of composed process models. Picus (cf. Sect. 13) can be used to search process models and corresponding performance data annotations for different analysis scenarios.

9.4 Related Tools

Another tool allowing model-driven performance analysis is PUMA [23,24]. It uses an exchange format similar to our TIPM but does not support feedback propagation from performance analysis. The SPMDA approach is another approach similar to MDPE [25].

10 Reuseware Composition Framework

The Reuseware Composition Framework¹⁰ [90,91,92] is a tool to develop component support for arbitrary EMF-based DSLs.

10.1 Background

Building software out of components has been a concern in software engineering for a long time. To enable component-based development, the languages used in

¹⁰ <http://www.reuseware.org/>

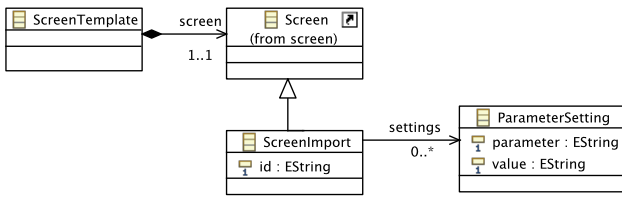


Figure 13 Metamodel extension for AppFlow import feature.

development usually have a certain built-in component support (e.g., concepts like classes, objects, or packages). The collection of these concepts is also called the *component model* of the language. Furthermore, languages need to incorporate a *composition language* that allows the definition of systems in terms of compositions of components (e.g., concepts like an import construct).

In the context of MDSD and DSM, new DSLs are constantly built and thus new component models and composition languages for these DSLs are required to retain the benefits of component-based development. Thus, we developed the Reuseware tool to specify component models and composition languages on the basis of Ecore-based metamodels. These specifications are used by Reuseware’s generic tooling. This way, a DSL can be enriched with component support without the need to manually implement composition tooling for each new DSL individually. Reuseware is itself developed model-driven using EMF for its internal metamodels, Fujaba’s story-driven modelling [15,93] to define composition semantics, and EMFText (cf. Sect. 5) to realise the editors for its specification languages (cf. [94] for details). Furthermore, it integrates OCL in its specification languages (cf. Sect. 7).

Reuseware is mainly applied during language application time and supports dynamic loading and reloading of component models and composition language specifications. Altering language implementation artefacts such as metamodels or syntax specifications is not required. The specifications in Reuseware are defined based on an Ecore metamodel and describe what components are in the corresponding DSL and how they are composed. Users of that DSL can utilise this component support to define and reuse components in the DSL. These are composed by a generic composition engine that merges models to a composed model that can be further processed by other tools. This way, composition is implemented as a preprocessor and has not to be handled during model interpretation or code generation. Note that Reuseware is a generic model composition tool and does by itself not specifically support metamodel composition for tool integration as discussed in Sect. 4. For this, where a preprocessor composition approach is not feasible, LanGems (Sect. 14) should be used.

```

1 START ScreenTemplate
2
3 RULES {
4   ScreenTemplate ::= "template" screen;
5   ScreenImport   ::= "import" id['<','>'] "as" name[]
6                   ("(" settings ("," settings)* ")")? ";"
7   ParameterSetting ::= parameter[] "=" value['"',"'"];
8 }

```

Listing 4 Syntax extension for AppFlow import feature.

10.2 Applied on Example

In the following, we demonstrate how import functionality is added to the AppFlow DSL which can be used to write reusable AppFlow components and compose them to new AppFlow models. Concretely, we add support to define templates for screens. These are screens with parameters defined independently of an AppFlow model that can be configured and reused in different AppFlow models. This is only one example of adding component support. Other modularity concepts for AppFlow or other DSLs can be realised with Reuseware. Examples of such include: reusable UML activity diagrams [90], aspects for UML and Java [90], and import mechanisms for rule languages [91].

In our example, we first extend the metamodel and the syntax of AppFlow with specific constructs to define templates and importing templates. As stated, Reuseware does not enforce such adjustments. Instead of extending a language syntax, one may also employ naming conventions or features such as annotations that are already included in the DSL. The interpretation of the new AppFlow constructs is defined afterwards with Reuseware. Figure 13 shows the extension of the AppFlow metamodel (cf. Fig. 2) with the metaclasses **ScreenTemplate**, **ScreenImport**, and **ParameterSetting**. The corresponding extension for the text syntax is shown in Listing 4 (cf. Listing 1).

With the extension, we introduce two new concepts into AppFlow. First, one can define screens independently of applications in template screens. Figure 14 shows such a template that uses the extended syntax, which provides the `<<parameter>>` notation to define parameters in texts. In the example, this is used in lines 2 and 4 to define the parameters `<<name>>` and `<<pwd>>` that are replaced by concrete texts displayed in front of the input fields when the template is reused in an AppFlow model.

Second, one can define imports and configurations of template screens inside an AppFlow model as for example shown in Fig. 15. There, the login template from Fig. 14 is imported and the parameters are bound to the values *Project Manager Name* and *Password*. Reuseware can now expand the import to the imported template screen and bind the parameter in this template by replacing the parameters with the corresponding values. In the case of the example, lines 30–36 of the model of Figure 3 are the result of the composition.

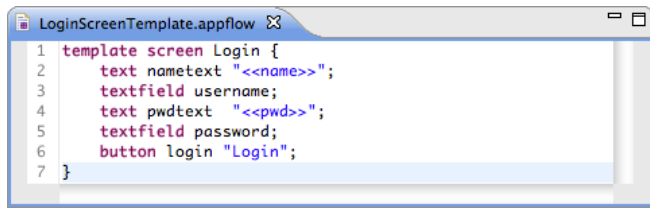


Figure 14 Template screen for a login dialogue.

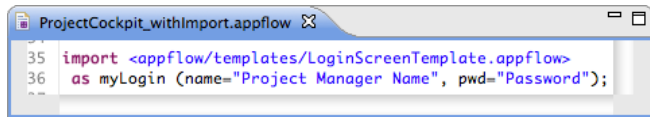


Figure 15 The login template (cf. Fig. 14) imported and configured in an AppFlow model.

In order to perform this composition of AppFlow models, Reuseware requires knowledge about the composition semantics of **ScreenTemplate** and **ScreenImport**. This is defined in terms of a component model and a composition language specification. The component model expresses that a **ScreenTemplate** is a reusable component and that strings are replaceable parameters if they contain a `<<parameter>>` value. This is done on the basis of the metamodel. For example, specifying **ScreenImport is hook** means that elements of type **ScreenImport** may be replaced during composition. The composition language defines that such a replacement is triggered by a **ScreenImport**. Concretely, it is defined that a **ScreenImport** is replaced with the **Screen** defined in a **ScreenTemplate** with the ID given in the **ScreenImport** to replace. This is expressed by `ScreenImport {id = $self.id.split('/')$}`. For details of component model and composition language specification consult [92].

The example in this section can only indicate the power of Reuseware's composition features. The component model and composition language specification formalisms give the tool developer capabilities to introduce support for arbitrary modularity concepts on top of EMF-based DSLs. This includes cross-cutting modularity concepts such as aspects. Component support can also span multiple DSLs in combination. For example, the language in which components are defined (for which a component model is specified) can be different from the composition language. For more examples refer to the webpage and to [95,96,97].

10.3 Role in DropBox

While none of the other DropBox tools directly requires Reuseware, it can be combined with all of them. An important property of model composition with Reuseware is that composed models do not include any composition related elements (screen imports in the example). Thus, the composition is performed transparently as a preprocessing step and other tools can work on the com-

posed models and do not have to know about composition related model elements. Dresden OCL (cf. Sect. 7) and MDPE (cf. Sect. 9) can thus do analysis on models composed with Reuseware without further adjustments. Model components can be searched for with Picus (cf. Sect. 13) and documented with DEFT (cf. Sect. 8). The FeatureMapper (cf. Sect. 12), can be used to create different variants of a composition. With the help of JaMoPP (cf. Sect. 15), Reuseware can compose Java source code.

10.4 Related Tools

The distinct properties of Reuseware are that it can be used with arbitrary DSLs, that it supports a large variety of different component types, and that it can be utilised without altering a DSL's implementation artefacts, if not possible or not desired. A number of tools and approaches that follow similar goals as Reuseware exist in Aspect-Oriented (AO) Software Development and related fields. Many of those are, however, restricted to specific languages or DSLs with specific properties. The approaches most closely related to Reuseware in terms of goals and functionality are: Kompose [26] that can also be used with arbitrary DSLs, but does not support the definition of new composition language constructs; GeKo [27], which is DSL-agnostic, but is oriented towards AO component types and does not support a separation of component model and composition language; GenAWeave [28], which can be applied to different DSLs, but is as well oriented towards AO component types; and MATA [29], which can conceptually be transferred to different DSLs, but is UML-oriented on the tooling side (which means that supporting new DSLs required implementation effort). For a detailed comparison of these and more related approaches consult [97, Chap. 10].

11 Refactory

Refactory¹¹ [98] is a framework for the specification of generic model refactorings which can be reused for arbitrary EMF-based languages.

11.1 Background

The term *refactoring* means the restructuring of existing code to improve its design while preserving its semantics [99]. This technique originates from the world of programming languages and, thus, is widely examined, accepted and used to reduce code complexity and coupling [100].

In MDSD, models instead of code are the primary artefacts used in the development process. In the previous sections, we already discussed tools to conveniently

¹¹ <http://www.modelrefactoring.org/>

specify syntax and semantics for custom modelling languages. As a result of this progress in metamodeling more and more modelling languages arise and improving the design of models becomes more and more important [98]. In that sense, refactorings can be invoked in the application phase of a DSL, since it facilitates restructuring of instances, or they contribute to the evolution phase of the DSL itself when applying refactorings to the metamodel. This motivated us to develop a generic refactoring framework for modelling languages.

Since we are faced with a large amount of modelling languages, model refactorings should be specified generically, which fosters reuse and ensures that they are not redefined again for each metamodel [101]. For example, the core transformation steps of a rename refactoring are the same in different metamodels if considered from a more abstract point of view.

For this purpose we introduced a novel approach of role-based generic model refactoring in [98]. The *role* concept originates from [102,103] where roles and collaborations between them are used to define different contexts for objects in a software system. This approach enables the reuse of the structure and the transformation steps by specifying the participating elements of a model refactoring as roles and defining the transformation based on that roles.

The approach was implemented in Refactory by introducing three new languages to specify and activate model refactorings in the metamodel of choice. Those languages are described in the following.

First, the refactoring designer has to define a generic refactoring by defining a *role model* which specifies a dedicated context for a certain model refactoring. The context contains the participating elements and the collaborations between them. These collaborations are the basis for navigating between the objects playing the corresponding roles while executing the transformation.

Second, the refactoring designer has to define a *refactoring specification* in which the concrete restructuring steps are specified. Those steps are only based on the previously defined roles. Thus, they are completely independent of the target metamodels. These two models constitute a generic model refactoring.

Third, the domain expert must map the defined roles and collaborations to concrete metaclasses and relations in a *role mapping*. Furthermore, such a mapping contains a name for this concrete model refactoring. Thus, in addition to the already mentioned application and evolution phases refactorings are executed in, they are made available in the implementation phase when role models are mapped to the metamodel of the DSL. Additionally, the formalism of role models enables Refactory to recommend possible role mappings to the domain expert which facilitates the process of specifying them manually [104].

Refactory is based on EMF and completely integrated into the Eclipse Language Toolkit [105]. As a result,

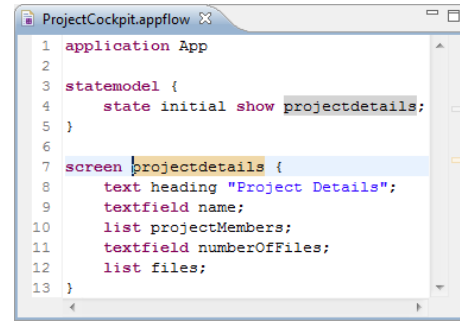


Figure 16 Example screen before refactoring.

model refactorings are natively embedded with Eclipse tooling and can be easily accessed by AppFlow users.

11.2 Applied on Example

As mentioned in Sect. 3, our AppFlow example instance contains a state model defining all states of the application and several screens referred by the states. A first model refactoring we implemented relates to the widgets being arranged on a screen. See, e.g., the screen **projectdetails** in Fig. 16. It contains a heading, two text fields and two lists. From a conceptual point of view, **heading**, **name** and **projectMembers** are more general information on this screen while **numberOfFiles** and **files** go into detail because they are intended to display information regarding the concrete content of the project repository. To visually separate general from detailed information, the domain expert could now manually create a new panel and move, e.g., the project files describing text field and list into the newly created panel.

We consider this screen restructuring a refactoring because widgets are only regrouped and no information is added or removed. In the context of AppFlow this modification is semantic preserving.

In consequence, we reused a model refactoring which was specified with Refactory for purposes when some model elements are intended to be moved into a new container element. This generic refactoring is called *Extract X with Reference Class*¹² and was inspired by the *Extract Method* refactoring known from programming languages. As mentioned above, the domain expert only has to map the roles of a generic refactoring to metaclasses in the desired metamodel. In the case of AppFlow, Listing 5 shows the corresponding mapping.

One can see that the concrete refactoring's name is *Encapsulate in Panel*, and, more important, that role **Extract** corresponds to any widget and the role **New-Container** corresponds to **Panel**. The new panel needs a name, which is why the role attribute **newName** is mapped to the attribute **name** of the metaclass **Panel**. Both roles **OrigContainer** and **ContainerContainer** are mapped

¹² Details about our generic model refactorings can be found under <http://www.modelrefactoring.org/catalogue/>.


```

1 ROLEMODEL MAPPING FOR <http://www.emftext.org/language/appflow>
2
3 "Encapsulate In Panel" maps <ExtractXwithReferenceClass> {
4
5     Extract := screenmodel.Widget;
6
7     NewContainer := widgets.Panel(newName -> name){
8         moved := compounds;
9     };
10
11     OrigContainer := screenmodel.Composite {
12         extracts := compounds;
13     };
14
15     ContainerContainer := screenmodel.Composite {
16         source := compounds;
17         target := compounds;
18     };
19 }

```

Listing 5 Role mapping *Encapsulate In Panel*.

to the metaclass **Composite** because only composites are intended to contain children. Since the name of the relation from **Composite** to its subwidgets is **compounds** (cf. Fig. 2), the collaborations **extracts**, **source** and **target** are mapped to it.

After specifying the mapping, a context menu entry for the refactoring is available in the generated AppFlow text editor. To invoke this refactoring, a user of the AppFlow language only needs to select the widgets intended to be moved and open the refactoring menu with a right-click on the selection. The result of *Encapsulate in Panel* for the file widgets can be seen in Fig. 17.

In addition to *Encapsulate in Panel* we defined two more model refactorings for the AppFlow language. First, the well-known *Rename* refactoring was applied to our language. This is a simple refactoring with large impact. If a nameable element is renamed, all references are updated automatically. With help of this refactoring the user has not to correct the invalid referring names manually anymore.

The last model refactoring we provided to the AppFlow language relates to a missing initial state which can be identified and signalled to the user as described in Sect. 7. In case of a missing initial state, the state model is not well-formed and the AppFlow user has to decide which state to choose as first reachable from an initial state. Although the behaviour of the state model will be modified, this restructuring is considered a refactoring because the user knows exactly beforehand which semantics are changed [106]. This refactoring is called *Create Initial State* and creates a state as predecessor of a selected state in the state model. Next to the newly created state a transition is added as well having the new initial state as source and the selected state as target.

All the described model refactorings can be invoked both in the generated text editor and the EMF generated tree editor. Furthermore, editor connectors for GMF editors, enabling refactorings for graphical editors, and for Xtext [7] editors exist.

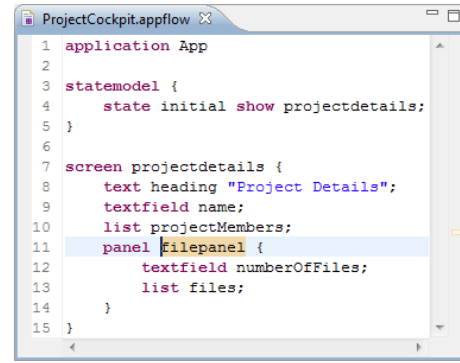


Figure 17 Example screen after refactoring.

11.3 Role in DropsBox

Refactory itself consists of three metamodels (role model, refactoring specification, role mapping) which are used to define one of the specification models as described in Sect. 11.1. To facilitate the definition process, a textual syntax was specified with EMFText (cf. Sect. 5) for each of the three metamodels. In return, to enable model refactorings in EMFText generated editors, Refactory provides an editor connector determining the model elements of the current selection on which a refactoring can be invoked.

Furthermore, Refactory was used to define some model refactorings for the model-based parser and printer JaMoPP (cf. Sect. 15). The benefit of this is that Java source code now can be handled as model and the *Java modeller* has not only a generated text editor available, but refactorings can be invoked in the same manner as known from the Eclipse Java Development Tools (JDT).

As indicated in a survey discussed in [107], refactoring support should be a mandatory feature in development environments for OCL. For this reason we implemented a couple of OCL refactorings with Refactory, such as renamings or extracting a variable from an expression and replacing all occurrences.

Finally, two model refactorings have been mapped to the metamodel of the FeatureMapper. First, the renaming of all feature model elements having a name can be invoked, and second, a constraint can be derived from selected features. For more details about the FeatureMapper, refer to Sect. 12.

11.4 Related Tools

In the scope of Eclipse-related refactoring tools being able to refactor models we refer to EWL as part of the Epsilon language family [30], EMF Refactor enabling model refactoring based on graph transformation [31], and the Operation Recorder supporting the creation of model refactorings by example [32]. In contrast to Refactory, none of these tools support generic definition of model refactorings. Another generic approach is introduced in [33]. The transformations are specified with

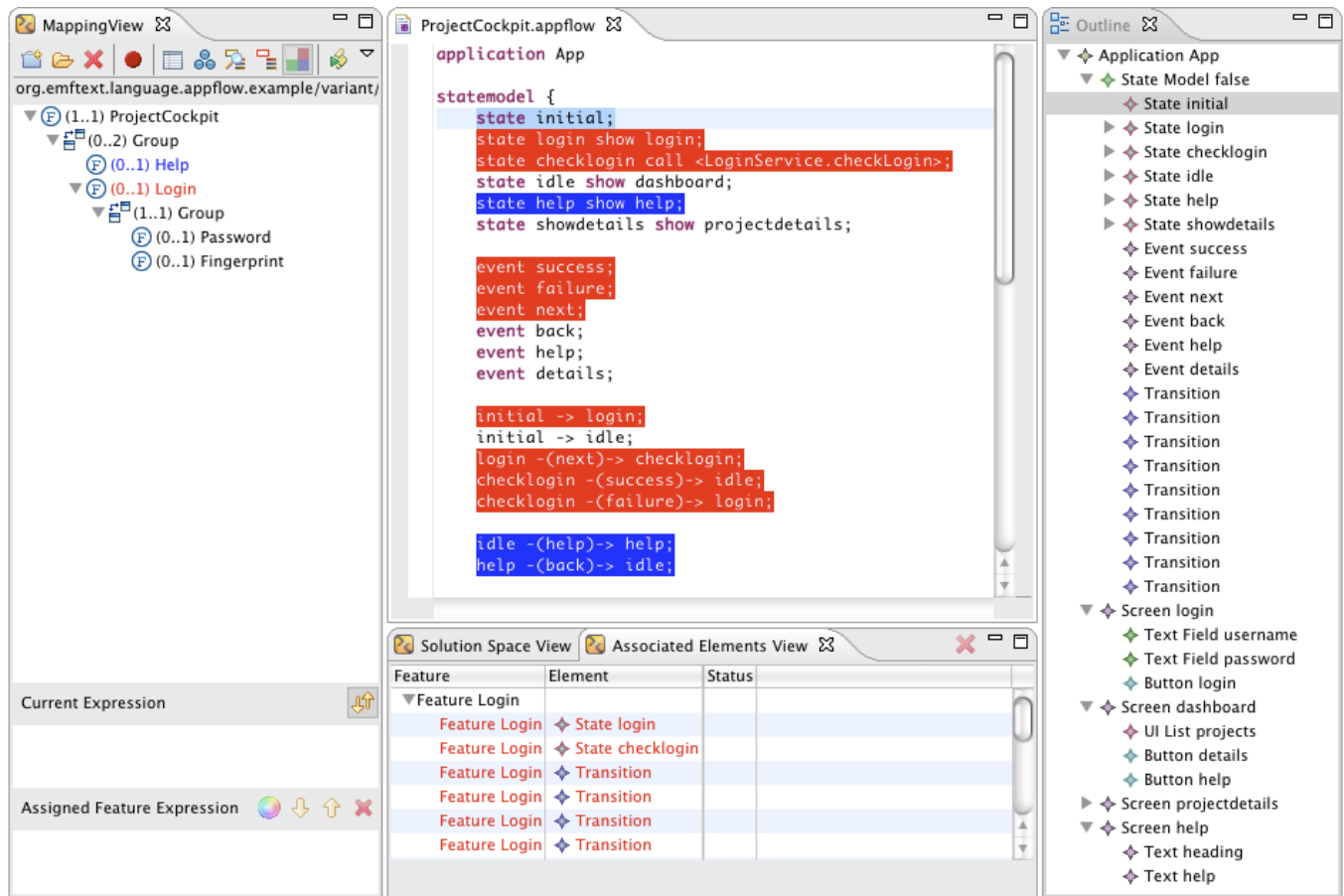


Figure 18 FeatureMapper with features mapped to parts of the example user interface specification.

Kermeta on top of a unifying metamodel called GenericMT. This approach is similar to ours but lacks flexibility because once the metamodel of the target DSL is mapped to GenericMT its not possible to map the same generic refactoring to another different structure of that metamodel. In contrast to this, Refactory allows for mapping one role model to different structures in the same metamodel and hence is more flexible. An in-depth discussion of related work can be found in [98].

12 FeatureMapper

FeatureMapper¹³ [108,109] is an Eclipse-based tool approach that combines MDSD and software product line engineering (SPLE) [110].

12.1 Background

A software product line (SPL) contains a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [110]. In

addition to the shared core assets, every member of an SPL has features that are specific to it and that are not shared by all other products of the SPL. To express this variability, the different features available in an SPL and their interdependencies are being described via variability modelling. Variability modelling resides in the *problem space* whereas the realisation of features is part of the *solution space* [111]. To instantiate products from an SPL, feature realisations in the solution space have to be configured according to the presence of the features in a *variant model*, that is, a concrete selection of features from a variability model that describes a product of the SPL. This requires a mapping between variability model features and solution space models or modelling artefacts that realise the features (or combinations of those).

FeatureMapper allows for mapping features from variability models to arbitrary modelling artefacts that are expressed by means of an Ecore-based language [5]. These languages include UML [112], DSLs defined using EMF, and textual languages that are described using EMF-Text (cf. Sect. 5). The mappings can be used to steer the product instantiation process by allowing the automatic removal of modelling artefacts that are not part of a selected variant from the final product being generated.

¹³ <http://featuremapper.org/>

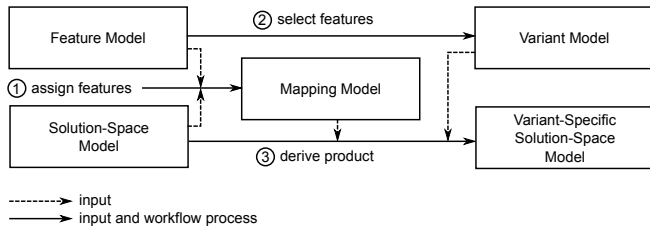


Figure 19 Workflow for defining an SPL and deriving a concrete product with FeatureMapper.

An overview of defining an SPL and deriving a concrete product in FeatureMapper is shown in Fig. 19. To associate features or logical combinations of features with modelling artefacts, the developer first selects the feature expression in FeatureMapper and the modelling artefacts in a modelling editor. Next, the feature expression is applied to the modelling artefacts via the FeatureMapper user interface (Step 1). During product derivation, this mapping is interpreted by a FeatureMapper transformation component. Depending on the result of evaluating the feature expression against the set of features selected in the variant (Step 2), the modelling elements are preserved or removed from the model (Step 3). Model elements that are not mapped to a specific feature expression are considered to be part of the core of the product line, that is, elements that exist in all members, and are always preserved. In addition to product derivation, the mappings are used for visualisation purposes [109].

12.2 Applied on Example

The example is specified using the AppFlow language. Since FeatureMapper directly supports mapping features from variability models to EMFText-based languages, we applied FeatureMapper on the AppFlow example to realise variability with respect to navigation and screen contents. For this, we defined a simple variability model in terms of a feature model [34, 113]. The feature model depicted in Fig. 20 specifies the **Help** feature and the **Login** feature as being optional. Furthermore, the **Login** feature consists of two alternative authentication mechanisms. We mapped the different authentication mechanisms to the implementation of the **JavaAction** that is associated with the **checklogin** state. That is, optional parts of the mapping address the AppFlow specification while the alternative parts address the Java implementation where FeatureMapper and JaMoPP (cf. Sect. 15) play nicely together. Figure 18 depicts an excerpt of the user interface specification of the **ProjectCockpit** example with the features mapped to the respective parts of the specification.

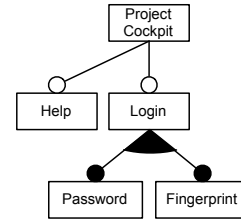


Figure 20 Feature model capturing the variability in the example SPL.

12.3 Role in DropsBox

FeatureMapper is a tool that works on top of the core services in DropsBox. It can be used with arbitrary EMF-based languages and, thus, can realise variability for instances of all languages in DropsBox; regardless of whether it is a metamodeling or a modelling language.

12.4 Related Tools

Related tools and approaches to FeatureMapper are the model templates by Czarnecki et al. [34] which provide mappings from features to UML artefacts by means of symbolic feature names in UML stereotypes. FeatureMapper goes beyond this work by supporting arbitrary EMF-based languages (including UML). CIDE by Kästner [35] was developed almost in parallel to FeatureMapper and provides means for mapping features to textual artefacts. In addition to CIDE, FeatureMapper abstracts from the concrete syntax and supports model-representations regardless whether those are textual or graphical models. VML* [36] by Zschaler et al. is a family of languages for variability management. In contrast to FeatureMapper, they use an operational approach where each feature is associated with a set of transformations that are executed on the models of the SPL.

13 Picus

Picus¹⁴ [114] is a faceted browsing interface for arbitrary models that allows for exploration of available models by complex filtering.

13.1 Background

Finding a suitable reusable component is still a challenging aspect of software reuse, in particular, when the components are models of various types that include everything from documents to source code. To address this problem, Picus uses the concepts of *faceted classification* and *faceted browsing*. Faceted classification is a flexible alternative to fixed catalogues. Instead of classifying an

¹⁴ <http://www.reuseware.org/picus/>

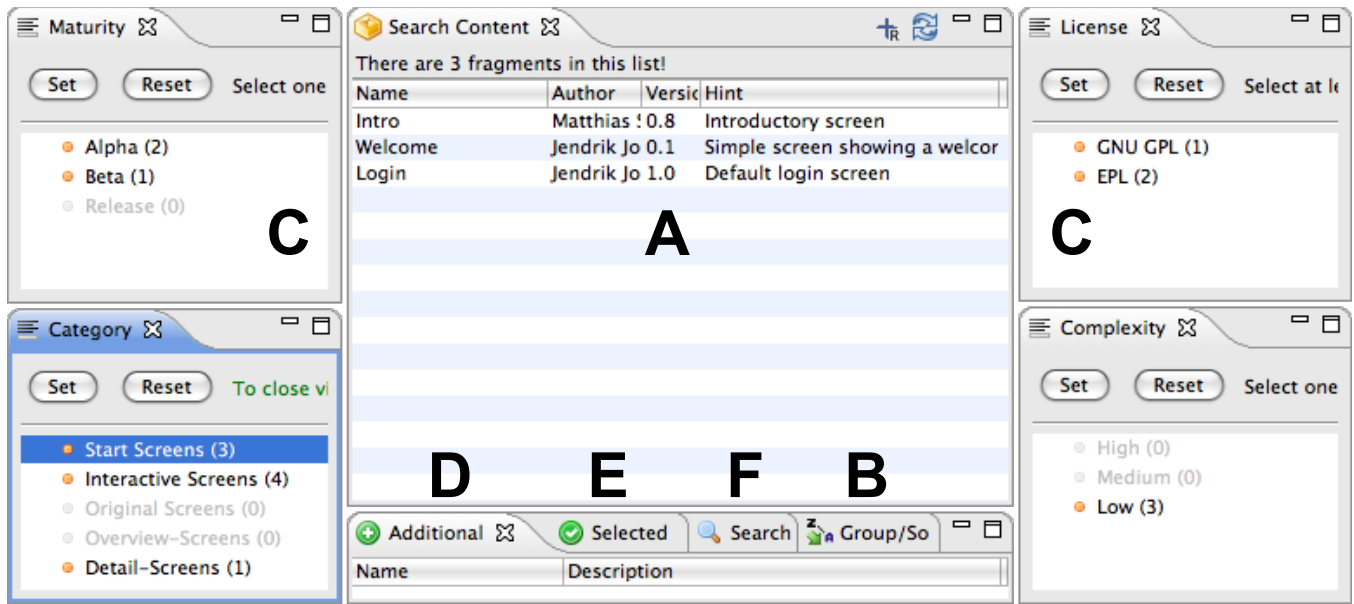


Figure 21 The faceted browser of Picus.

item into one class in a hierarchy of predefined classes, the item is classified according to multiple smaller taxonomies [115]. Faceted browsing [116] is a user interface paradigm that builds on this classification principle. Its main benefits are: (1) The user can search by restricting (and relaxing) facets at any time and in arbitrary order. This allows him to browse according to his or her own navigation path. (2) The user will never get empty result sets as with normal filtering and searching. This is impossible by construction of the filtering interface and contributes to the users browsing experience. (3) The user does not require knowledge in advance about the things that he wants to browse, but can explore completely unknown datasets. In the last years the popularity of faceted browsing was pushed by web applications (mostly e-commerce) [117] and generic web-based browsers emerged (e.g., Flamenco [118] or Exhibit [119]).

Picus was developed with the intent to use it in combination with Reuseware (cf. Sect. 10) to browse a repository for reusable model components. In the context of the MODELPLEX project, it was used in this combination in a case study defined by Telefónica R&D where faceted classification and browsing was recognised as a useful approach. More information about Picus and the case study can be found in [114].

Picus supports three tasks related to faceted classification and browsing: The definition of facets, the classification of components that can be done manually or rule-based, and finally the actual browsing. The definition of new facets is a task that relates to language implementation, since new domain-specific facets may be needed for specific DSLs. Classification is done during language application by classifying models using facet values. Alternatively, one can add derivation rules (formulated in OCL; cf. Sect. 7) to facets which perform a classification

```

1 if Widget.allInstances().size() < 10 then
2   'Low'
3 else if Widget.allInstances().size() < 30 then
4   'Medium'
5 else
6   'High'
7 endif endif

```

Listing 6 OCL derivation rule for the facet *Complexity*.

automatically by inspecting the contents of the model to classify. Once models are classified, one can browse for existing models during the language application phase.

13.2 Applied on Example

We utilise Picus to browse for AppFlow models. In particular, we classify and search in a set of template screens, as introduced in Sect. 10.2 (Reuseware example).

We started by defining a set of template screens and classifying them. For that, we used the following general facets that are applicable to any kind of model: *Maturity* and *License*. Furthermore, we defined, two domain-specific facets for AppFlow models: *Category* and *Complexity*. For the facet *Complexity*, a derivation rule was defined with OCL. Facet value derivation rules automatically derive the classification for a model by inspecting it. The rule for the facet *Complexity* is shown in Listing 6. The rule counts the number of widgets in a screen template and automatically classifies that screen template with one of the facet values *Low*, *Medium*, or *High*.

Figure 21 shows the faceted browsing perspective of Picus that is integrated into Eclipse. The browser's features range from special widgets for presenting facets, over a free-text search to features such as grouping and sorting. The important parts of the browser are marked

in Fig. 21 and include the main functionality of a faceted browser. These parts are the result view (A), a grouping and sorting facility for the result view’s entries (B), as well as widgets to present available facets and their values (C). As there might be more than six facets available, a separate view lists other available facets (D). While the user selects facets and values to perform zoom-in and zoom-out steps with (C) and (D), the current search query is shown in another view (E). Finally, a search view gives the opportunity to perform a free-text search over available facets and classifications (F). These features can be used to search for arbitrary models.

In the example search of Fig. 21, the AppFlow model *Login* (cf. Fig. 14) is part of the result set (A). The domain-expert can now use this result and import the such found model component in a new AppFlow model as discussed in Sect. 10.2 (Reuseware example).

13.3 Role in DropsBox

Picus, as a browser for arbitrary models, can be used in combination with all DropsBox tools to find models that are to be processed by one of the other tools. As such, it can be helpful during language application and development, since it cannot only be used to search for models, but also for Ecore metamodels, EMFText syntax specifications (cf. Sect. 5), OCL expressions (cf. Sect. 7), Reuseware component model or composition language specifications (cf. Sect. 10), and others. Furthermore, the rules for automated classification are defined in OCL (cf. Sect. 7) and specific integration with Reuseware (cf. Sect. 10), to support drag and drop of search results from the result view into model compositions, is provided.

13.4 Related Tools

Although component libraries, such as CORBA [120] and UDDI [121], as well as libraries for specific modelling languages exist, we are not aware of any faceted component browser in the context of DSM that is combinable with arbitrary DSLs and offers comparable filtering and browsing capabilities. The support of the faceted browsing paradigm is the main distinguishing criteria here. MoDisco [37] offers a generic model editor that allows the user browsing a model defined in an arbitrary DSL and defining custom queries comparable to facet derivation rules. However, this editor is designed to browse a single large model instead of libraries of models. For a comparison of work concerning libraries with facet support in other software engineering disciplines, refer to [114].

14 LanGems

LanGems¹⁵ [122] provides a metamodeling approach supporting the definition of self-contained and reusable

¹⁵ <http://www.langems.org/>

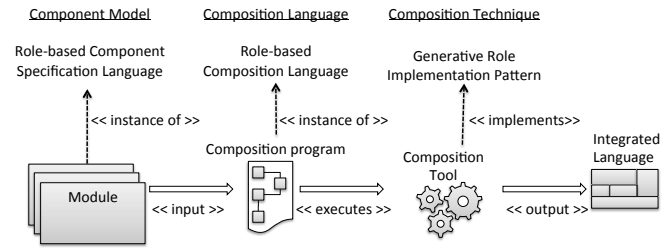


Figure 22 LanGems – a role-based language composition system.

language components and their composition in easily evolvable, adaptable and extensible language families.

14.1 Background

A language family denotes a set of related languages that share some common syntactic and/or semantics concepts but also differ in certain elements. The development of language families is motivated by the potential for reuse among family members and the need to adapt and extend languages to different application domains and contexts. We discussed and evaluated the potential of language families and motivated modular development of DSLs [123], ontology languages [124], OCL [122], and metamodel-based tool integration [125].

Based on our experiences in invasive software composition (ISC) [126] we investigated a number of existing language engineering approaches with respect to their applicability in modular language development [123]. Although various approaches can be found that support modularity in language syntax and semantics, it turned out that current inheritance-based composition mechanisms restrain the independence and, thus, reusability of individual language components [122]. Furthermore, we identified a lack of systematic support for component adaptation during language composition [122]. To address these challenges we suggested to consider the language’s abstract syntax—i.e., the language metamodel—as the primary artefact in a dedicated language composition system.

A composition system can be defined in terms of a *component model*, a *composition language* and a *composition technique* [126]. To provide these three elements LanGems extends the EMF metamodeling infrastructure with a role-based [102,103] language composition system. Role-based modelling evolved in the domain of software design as a natural extension of class-based modelling. It focuses on the collaboration of objects that play context-dependent roles to implement specific system concerns. Such collaborations form individual components of reuse that are superimposed to derive a system implementation [127]. This motivates the application of role-based collaborations as the component model of our language composition system and a role-based composition language to describe the superimposition

of several role components (cf. Fig. 22). As composition technique we apply a generative role-implementation pattern that transforms a role-based language specification to an integrated language implementation.

Increased reuse of language components and an enhanced flexibility during their composition is expected to enable a more efficient realisation of languages. In this section, we demonstrate the application of language composition to AppFlow and discuss benefits with respect to language evolution, adaptation, and extension.

14.2 Applied on Example

The packaged structure for the AppFlow metamodel presented in Fig. 2 already indicates a first motivation to apply a compositional language engineering approach to implement AppFlow as a language family. Subpackages like **statemodel** or **screenmodel** introduce sublanguages which would benefit from independent evolution, and flexible adaptation and extension for different application contexts. Consider, for instance, a logging protocol specification language that relies on the state-based abstraction, but does not necessitate a specification of user interface elements, or a user-interface designer that applies user interface specifications independently of a state-based application flow. However, such independent evolution and reuse of components is currently hindered by their tight and strong coupling. In the following we will exemplify how the application of our language composition system helps to tackle these issues by design. We discuss how AppFlow is modularised in role-based language components and how role composition helps component adaptation and reuse.

Figure 23 depicts role-based language components derived from the AppFlow language. Each module corresponds to a package of the AppFlow language. Besides natural classes, as found in the plain metamodel, special role classes are represented by the rounded rectangles. Such role classes specify the expected interface of a language component, i.e., parts of the component that are expected to be bound and specialised by classes found in other language components. Furthermore, each role class can define role features and operations (in the body of the role class) that contribute to the role interface.

For example the role class **Action** introduces no role features but the role operation **execute()**. This role class prepares the statemodel for extension with concrete actions that may be defined in another language component. Other role classes are **Widget** preparing the widgets used in the **screenmodel** for extension, or **Effect** allowing other language components to refine the specification of results for clicks on a **Button**.

In contrast to the packages in the AppFlow metamodel there are no connections across language components which makes them self-contained by design and helps their independent evolution and reuse. Besides its

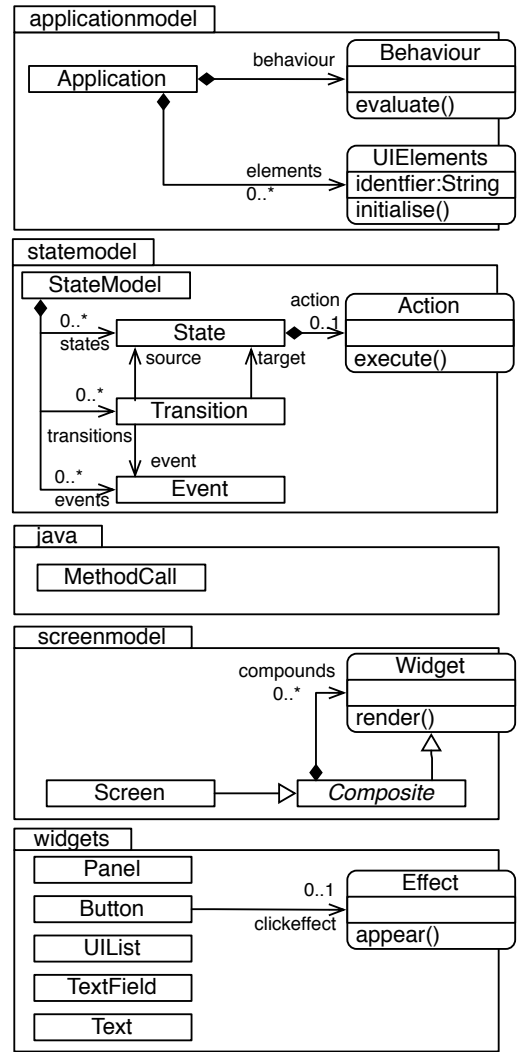


Figure 23 Role-based language components for AppFlow.

metamodel, defined in terms of a role model, each language component may contribute a concrete syntax and semantics specification solely defined against the role model. For a more detailed discussion of syntax and semantics specification in LanGems we refer to [122].

Next, we want to derive an integrated language from the role-based language components defined in the previous section. Therefore, we specify a language composition program using role bindings between the components. Each role binding connects natural classes with the role they play in another language component. Figure 24 depicts the role bindings necessary to connect the introduced language components to the AppFlow language.

Here, a **Statemodel** plays the role of a **Behaviour-Specification** in the **applicationmodel** component, the **Action** role in **statemodel** is played by **Screens** from a **screenmodel** or **MethodCalls** from **java**, and the **Effect** of a button click may be an **Event** in the **statemodel**. In these role bindings it has to be spec-

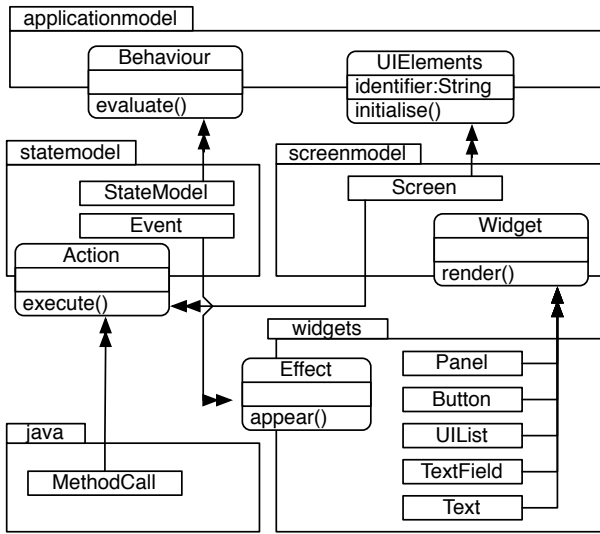


Figure 24 Composition program for AppFlow.

```

1 Screen plays UIElement {
2   identifier : name; // bind structure
3   initialise() : setupScreenComponents(); // bind semantics
4 }

```

Listing 7 Example of role operation binding.

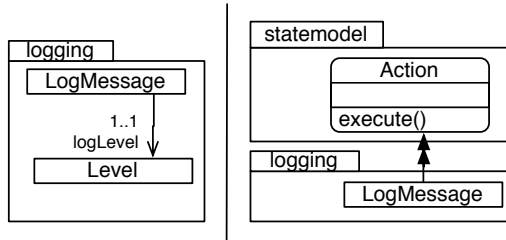


Figure 25 Logging component and composition of logging protocol specification language.

ified how role players implement the role features and operations of the role they are bound to.

The binding of **Screen** to **UIElement** is exemplified in Listing 7 by mapping screen feature **name** to feature **identifier** (line 2). Also, **setupScreenComponents()** is mapped to the operation **initialise()** from **UIElement**. Thus, role bindings do not only connect language components but also provide flexible means to adapt and integrate their structure and semantics.

The AppFlow language can be considered as a particular language variant that can be derived from the language family formed by these components. Figure 25 introduced a language component to specify log messages and shows how to derive a language variant for the statemodel-based specification of a logging protocol.

14.3 Role in DropsBox

The LanGems language composition system is tightly integrated with the EMF. Consequently, it can be com-

bined with other DropsBox tools applied by tool developers during language engineering. For instance EMF-Text was used in various applications to enable the composition of concrete syntax for language components. Tools like Dresden OCL (cf. Sect. 7) or JastEMF (cf. Sect. 6) can be combined with LanGems to modularly define well-formedness rules or static semantics of language components. A combination of the FeatureMapper and LanGems during language engineering eases the specification and customisation of language families. Here feature models could be used to specify supported language variants and mapping them to the corresponding fragments of a language composition program.

14.4 Related Tools

LanGems aims at contributing to modularity in language metamodeling. It, therefore, is related to Eclipse-based metamodeling tools like EMF [5], KM3 [38] or Ker-meta [39]. In addition, there are further implementations of MOF-based metamodeling tools like Netbeans-MDR [40] or MOFLON [41] and proprietary tools with metamodeling capabilities like JetBrains MPS [42], Meta-Edit+ [43] or Microsoft OSLO [44]. All these approaches use packages for modularisation and package import as means for connecting language modules. As discussed in detail in [122,123] this induced issues with information hiding and module reusability. The tool HIVE introduced in [45] proposes an alternative modularisation of languages using slices and roles. For a comparison to LanGems we also refer to [122].

15 JaMoPP

JaMoPP (Java Model Parser and Printer)¹⁶ [128] is an EMF-based implementation of the syntax and the static semantics of the Java programming language. It fosters the reuse of the Java language when designing new DSLs. Also, JaMoPP targets language evolution when existing Java programs are analysed and DSL models are extracted.

15.1 Background

Java is an object-oriented programming language that is widely used both in academia and industry. To employ Java in the context of MDSD, one must bridge the gap between modelling and programming. Typically, this is performed by generating code from models to obtain an executable specification of the system at hand. The fact that models were introduced to provide higher abstractions than what is usually found in general-purpose languages (GPLs) often caused the perception that an

¹⁶ <http://www.jamopp.org/>

explicit distinction between the two types of languages is required. We think this perception is not only wrong, but leads to several problems. First, modelling tools cannot be applied to programs. Second, well-known research results in the area of programming languages are reinvented by the modelling community. Overcoming these drawbacks and making Java a first-class citizen in the modelling world motivated the idea of building JaMoPP.

The main motivation to build JaMoPP was to allow for the application of EMF-based tools to Java programs. In particular, we were interested in composing programs with Reuseware (cf. Sect. 10), mapping features of software product lines to code fragments (cf. Sect. 12), and to investigate the synchronisation between generated code and models. But, JaMoPP was not only built with these three application areas in mind. It was clear, that applying EMF-based tools to Java programs would yield much higher potential. To name a few potential applications, syntactically safe code generation was enabled by JaMoPP [129], language extensions could be built, and model transformations could be applied.

Also, besides the direct applications that were enabled by JaMoPP, the project turned out to be a crucial test for EMFText. The complexity of the Java language both from a syntactical and from a semantical point of view made many shortcomings of EMFText obvious. Thus, EMFText did benefit to a large extent from the development of JaMoPP, in particular with respect to stability, performance and scalability.

The main building blocks of JaMoPP are its Java metamodel, the concrete syntax specification based on EMFText (cf. Sect. 5), a model extractor for `.class` files and additional tooling to integrate JaMoPP with the Eclipse JDT. The metamodel consists of roughly 230 metaclasses in 18 packages. The syntax specification contains about 150 syntax rules—one for each concrete metaclass. Based on the tooling generated from this syntax specification by EMFText, JaMoPP can convert Java source code to models and also print source code from models. The model extractor for `.class` files is needed to resolve references to libraries where source code is not available. Finally, the JDT integration connects JaMoPP smoothly with Eclipse without worrying about how to configure the classpath. In summary, JaMoPP lifts Java to the level of modelling languages and enables tool developers to treat Java just like any other modelling language that is based on EMF.

15.2 Applied on Example

Within AppFlow, we used JaMoPP to connect application models specified with AppFlow with custom Java code. In particular, a specific action (**JavaAction**) was introduced that can reference Java methods. When the application model schedules a **JavaAction**, the referenced method is executed. By introducing such an escape mechanism, parts of applications that cannot be

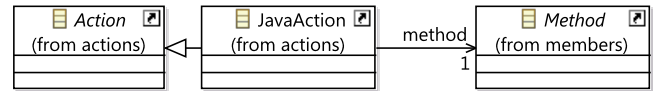


Figure 26 Metamodel for JavaActions.

```

1 statemodel {
2   // ...
3   state checklogin call <org.example.LoginService.checkLogin>;
4   // ...
5 }
  
```

Listing 8 Example instance of JavaAction.

modelled, or that are easier to program in Java, can be safely embedded in the descriptive application model.

To realise **JavaActions**, the metamodel of AppFlow contains a class that extends the abstract superclass for all actions—**Action**—and that holds a reference to the type **Method** from the JaMoPP metamodel. The definition for this class is shown in Fig. 26.

The metamodel of the Java language that is provided by JaMoPP enables one to establish references from modelling languages to Java programs. Arbitrary parts of Java programs can be referenced and thereby integrated into models in a type-safe way.

To resolve the reference `method` (i.e., to look up the actual Java method in the source code), a custom resolver class is needed. This class uses the JaMoPP infrastructure to find the referenced class by its fully qualified name and to check whether the class contains a method with the correct name. The resolver class needs to be coded manually, because neither the metamodel nor the concrete syntax definition of AppFlow provide information on how to find methods. Basically, the resolver class extracts the fully qualified class name and the name of the requested method from the dot notation, uses JaMoPP to look up the respective class and iterates over the members of the class to find a matching method.

An example instance of an action that refers to a Java method is shown in Listing 8. Here, the `checkLogin` method in class `LoginService` will be called whenever the application reaches the `checklogin` state.

Since JaMoPP can access Java code as any other model, we can ensure at development time that the referenced method does not only have the correct name, but also the correct signature (i.e., parameter types and return type). This way, type errors can be detected early (i.e., before code generation or interpretation).

In the context of AppFlow, JaMoPP can only be used to a limited extent. In general, JaMoPP provides more opportunities for application in MDSD and DSM. First, AppFlow uses an interpreter to run modelled applications. If we would employ code generation to derive applications from application models, we could use JaMoPP to ensure the syntactical correctness of generated applications [129]. This way more guarantees about

the result of code generation can be given, which is not possible with model to text transformation engines, but essential if code templates are used by third parties.

Second, JaMoPP enabled the embedding of AppFlow models directly in Java programs. To perform such an extension of the Java language, one can simply import the concrete syntax definition of Java and augment it with additional rules (e.g., rules of the AppFlow language). Then, AppFlow models can be directly embedded in Java classes. Examples for such extensions can be found in the EMFText Syntax Zoo [56]. To compile extended programs with ordinary Java compilers, the additional concepts must be translated to plain Java (e.g., using a model transformation). The examples in the Syntax Zoo are accompanied by such transformations.

Of course, language integration can also be realised in the opposite direction, that is to reuse Java concepts and syntax in AppFlow models. A good candidate for such an integration is to reuse Java expressions to model guards for transitions in AppFlow state models.

Third, JaMoPP can be used to reverse engineer existing applications [130] in order to derive AppFlow models. One can obtain a model-based representation of the existing Java code and search for code fragments that can be replaced by AppFlow models.

15.3 Role in DropsBox

JaMoPP's role in the DropsBox toolbox is to connect modelling tools to the Java language. JaMoPP bridges modelling and programming in the sense that it can treat Java programs as models. Thus, modelling languages can refer to Java programs in a type safe manner, modelling languages can reuse Java concepts by importing the Java metamodel, Java's concrete syntax can be embedded in other languages and other DSLs can be directly embedded in Java.

Besides the direct opportunities that are enabled by representing Java programs as EMF models, we are convinced that JaMoPP has great potential as a platform for a variety of empirical experiments. Given the huge amount of Java source code that is freely available on the web, the number of Java models that can be obtained with JaMoPP probably outnumbers the instances of any other language built on top of EMF. Having a formal representation of all this code, is the basis to perform comprehensive analysis with respect to the usage of the Java language. For example, we published results about the frequency in which proposed extensions to the Java language (e.g., Closures) are actually used in Java programs [131]. The amount and the size of available Java models can also be a significant test setting for modelling tools. Other tools can benefit from such stress tests similar to the process EMFText went through during the development of JaMoPP.

In addition, eJava¹⁷—a language that is built on top of JaMoPP—allows the augmentation of EMF metamodels with operation bodies. This way, hand-written code can be cleanly separated from generated parts. We used eJava to implement other DropsBox tools (e.g., EMFText and Refactory).

15.4 Related Tools

Extracting models from Java programs is also targeted by other approaches. MoDisco [37] and Spoon [46] use the Eclipse JDT tooling to retrieve syntax trees from Java programs and do also provide metamodels for Java. However, both approaches lack extensibility and reuse functionality that is enabled by the syntax specification that is available from JaMoPP. For further information about JaMoPP please refer to [128] and [132].

16 Lessons Learned

In the previous sections we illustrated our perspective on current and future DSL development with DropsBox. The development of DropsBox was driven by our research and is thus deeply integrated with other research activities. Our experience in such tool building in the academic context is discussed in this section.

Many researchers in the software engineering community build tools. However, one can pursue the development of such a tool to different degrees of maturity. First, one can build very limited *prototypes* that show the feasibility of some idea or algorithm. The effort associated with building a prototype is relatively low. It can be achieved by one or few developers in a rather short time and typically in the context of a single research project. Usually the user audience of such prototypes is limited to the tool developers themselves. This is perfectly fine, since the objective of the prototype is to test whether some idea can be implemented and whether some theoretical result can be confirmed. Most of the DropsBox tools discussed earlier (e.g., DEFT, LanGems, Picus, Refactory) represent or evolved from such prototypes. For these examples, demonstrating the research idea with an prototypical tool helped in gaining interest and convincing colleagues, students and reviewers.

Second, one can build *demonstrators* which have a slightly broader audience in the sense that they are designed to be presented to a second party (e.g., other researchers). This kind of tool requires more effort, as one must implement means to present the aim of the tools in a comprehensible manner. Usually this involves building a user interface and providing some example that is processed by the tool. Thus, the development of demonstrators requires more resources and time. The set of

¹⁷ <http://www.emftext.org/language/ejava/>

users may not only be restricted to the developers themselves or a single project, which introduces some additional effort for documentation and training. With tools like FeatureMapper, JaMoPP, JastEMF, Reuseware, or MDPE we learned that producing such documentation and training material is often worth the effort. It supports the demonstration of our approaches in tutorials and workshops. In return, the newly gained users and contributors foster the evolution and improvement of the approaches implemented on our tools.

Third, one can build *initial products*. Such tools explicitly target other stakeholders as users. These can be students, fellow researchers, industrial project partners, or any other party who is interested in using the tools. In contrast to the previous category, products require even more effort, since one must provide a degree of maturity that allows new users getting familiar with the tool and using it on their own, given a small amount of help. Typically, products at such an *initial* or *repeatable* level [133] are extensively documented and quality-assured by testing with satisfactory results. As a consequence, there is a lot of development effort associated not only with the realisation of the initial idea, but also with adjacent tooling, maintenance, bug fixings and community building. This typically involves multiple developers, external partners, and scheduling of resources from multiple projects over longer periods of time. Such investments need a strong motivation, but on the other hand can lead to good impact in both industry and academia. The laborious development of EMFText paid off by motivating, helping, and enabling the development of other DropsBox tools and also attained interest from external users in academia and industry. During the long development of the Dresden OCL we experienced growing interest from academia and practise, too. We were influenced by several versions of the OMG OCL standard and also contributed to the standard. We can report success stories on different practical and academic applications.

We think that each of the three categories has its own right to live, but we are also convinced, that researchers can sometimes benefit from going the extra mile to push some idea to an initial product, rather than remaining with a prototype or demonstrator. Over the last years, we have made very positive experiences, in particular with the DropsBox tools that are most mature.

We are very well aware that countless hours have been spent on development, fixing bugs and writing documentation. Still, when looking at the positive implications of all this effort, we felt that these hours were well spent. However, this is of course a very subjective impression, which can hardly be proven. In the following, we discuss different aspects that contribute to our impression. To obtain an overview of these aspects, consider Fig. 27.

We are convinced that different aspects of research, illustrated as circles in Fig. 27, always play together for successful and sustainable software engineering re-

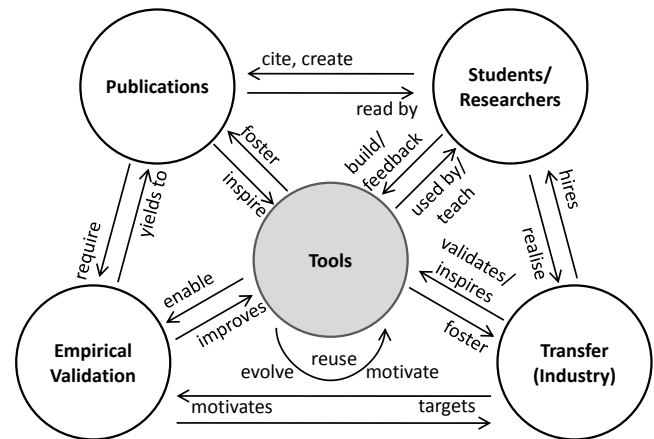


Figure 27 Aspects of tool building.

search. Therefore, we believe it to be dangerous ignoring any of these aspects, and tool building in particular. Through the influences between the aspects (arrows in Fig. 27) they support and foster each other. From our observation, tool building can exploit these influences to have substantial impact on the other research aspects and therewith multiply their results both in quality and quantity. In the following, we summarise our experiences of the influence tool building has on the other aspects.

Scientific publications and career A researcher is often judged by the quality and number of his or her publications. Although tools can be as visible as publications and the size of a user community is a good indicator for quality and impact of a tool, this does not necessarily help a software engineering researcher’s career directly. In many cases, this leads to the assumption that building tools beyond the prototype level is not worth the effort when pursuing an academic career and that the time is better spent writing publications.

In our experience, this is often too shortsighted. We rather experienced that the additional time spent on driving prototypes towards products is time that is repaid by allowing us writing better quality and more cited publications, as well as writing those in a shorter amount of time. In the long run, we were thus able to produce both initial products and high quality publications. Furthermore, tools like Dresden OCL and EMFText, which are in initial product state, are now used by other research groups. This leads to an increased visibility in the community. The tools act as evidence that high quality work is performed in our group. This all integrates us more into the corresponding research communities which also has positive feedback on publishing on the long run.

Concretely, we experienced three major factors where tool building improved publishing in our group. First, as we will further discuss below, tool building led to better cooperation between researchers in our group, because good working tools convinced researchers in our group that they benefit from each others’ work. This enables

joint effort which eventually leads to the writing of publications with a group of people that already works well together as a team. Second, writing a publication that is directly based on a mature tool, is much more straightforward, because one does exactly know what works and what does not. In prototypes, often certain functionality that is not directly in the core of one's research are not implemented completely. Upon writing, one sometimes has to navigate around these issues, unable to precisely describe them, which is time consuming. Third, having tools working well, allowed us doing evaluations (cf. empirical validation below) quickly, which could then be included in the publications to raise their quality.

Although having built a tool of product quality might not be of much help when applying for a position in academia, one should not forget that software engineering is a practical discipline. Thus, one should always question the practical applicability of one's research results. Doing that by building software is a very good opportunity. Thus, from our experience, it does not hurt PhD students to do concrete engineering of software to nurse and improve their own engineering skills.

Research community and teaching Our group has long-term experiences in building tools in an academic environment and in using them in research as well as in teaching. Often bachelor and master students design and implement tools in their theses, and by this means contribute to the validation of research results of PhD students. Not surprisingly, we learned that a strong motivation of students is a big success factor in building tools with high software quality. Motivation can be developed in particular by requests and feedback from external academic or industrial partners/customers and by creation of a productive team atmosphere. A good reputation in the research community improves this effect.

Furthermore, students and researchers are typically the first users of the developed tools. Consequently, we use our own tools both in research and teaching to get valuable feedback for further development activities. By doing so, external partners accept the high qualification of our alumni. This regards not only the advanced knowledge in MDSD and DSM but also the skills in professional software development in general.

Empirical validation In recent years, empirical studies gained more and more importance for both the scientific and practical evaluation of methods and approaches in software engineering [134]. The results of empirical validations are interesting for several reasons. They provide valuable feedback for the evolution of tools and engineering approaches. They provide evidence for the evaluation of research results in scientific publications. Finally, empirical validations are an important foundation for research targeting transfer of research results to practise.

The maturity of a tool implementing or supporting a given engineering approach is a key factor for such

empirical studies. First, it influences the quality of the evaluation results. An insufficient tool implementation may mask the approaches' benefits or obfuscate its weaknesses. Second, a higher number of external users eases the acquisition of empirical data. We experienced that different maturity levels enable different models of empirical validation [135]. While prototypes are typically used to assert the feasibility of a new technology, more advanced validations like replicated experiments, case studies, or field studies require higher maturity levels. This need for reliable validation results in publications, for tool evolution and in technology transfer, contributes to our motivation of building mature tools.

Technology transfer and industrial partners One important goal of applied research is to transfer new ideas and methods to the industry. Such transfer can be realised in different ways. First, students carry their knowledge to companies when they finish their studies. Second, industrial partners take part in research projects, which is also an opportunity to sell new research results to an industrial audience. Third, new enterprises can be created to enable the commercial exploitation of new methods. Fourth, providing tools implementing novel engineering approaches freely or open source allows such transfer of technology.

We think that all four categories of technology transfer do rely to some extent on the maturity of tools built by researchers. In teaching, we can hardly convince students of some new technology if they cannot experiment with the respective tools. In the best case, they will go off to their employers and tell them about this nice idea which is far from being useful in practise. The same can be observed when trying to convince industrial partners directly. If one cannot solve the problems of the partner for reasons of tool immaturity, it is usually hard to convince them of the idea behind the tool.

Founding a new enterprise based on a tool, which is too far away from deserving the label product, will also eventually fail. If one cannot create something that can be sold within a reasonable time span, the new enterprise will run out of money and vanish. Finally, providing tools free of charge or open source does not make any sense, until a certain level of maturity is reached. A user community will only establish if the provided tool is actually usable. Once this stage is reached, there is a good chance that other people will get involved, but from our experience this does not happen before at least the state of an initial product is reached.

Building multiple related tools Developing various related tools differs from building an individual tool in many ways. First, existing tools can be used for building other tools. This has the benefit of detecting bugs and design issues in existing tools at an early stage and getting feedback and enhancement requests from within the group. It is the *eating your own dog food* principle,

which fosters usability and maturity of new tools. Second, if multiple related tools are built, it does not hurt too much if the development of one tool is considered to be not worthwhile to be continued. It has the evolutionary aspect of investing more resources in promising ideas and tools and letting the others vanish. Third, we observed that building related tools provokes interesting dynamics within the development groups, both with regard to mutual motivation and increased interaction. The latter is an important factor and we observed that building those different tools led to effects, where people started working together regardless of their research project affiliation or their individual research interests.

17 Conclusion

In this paper we presented the Dresden Open Software Toolbox (DropsBox), a platform consisting of eleven tools for DSL development and application. The DropsBox tools support various activities in the life cycle of a DSL. All DropsBox tools are based on EMF which is used as modelling tool during the analysis and design of a DSL. Furthermore, EMF is the integration platform between the DropsBox tools and allows for seamless interoperability between and integration of the different life-cycle phases of a DSL.

The DropsBox provides an integrated tool set for language development supporting language design, implementation, deployment and evolution. In language application DropsBox tools contribute features that were previously only available for general-purpose modelling or programming languages. Although some tools can be exchanged by related tools, their tight integration in a toolbox offers a unique environment and comprehensible approach for the DSL life cycle. We demonstrated these features and their benefits using a running and tested example.

Summarising our DropsBox research, we discussed the lessons learned in building and integrating language engineering tools. Our experiences are that building mature tools in the context of research needs careful management of resources, but fosters other research activities. In particular, we explained, how tool building serves the collaborations of researchers within our group and the collaboration of our group with others from the DSL engineering and MDSD research communities as well as industry.

The DropsBox provides a unique set of capabilities among the modelling and metamodeling toolboxes (also called language workbenches) we are aware of. We roughly distinguish them into EMF-related toolboxes and toolboxes that use another integration platform. The best known EMF-related toolboxes are the Eclipse Model Development Tools (MDT),¹⁸ Epsilon¹⁹ and the former

openArchitectureWare (oAW) consisting of tools like Xtext, Xtend, and Xpand.²⁰ Other toolboxes use proprietary integration platforms to bundle a number of tools for DSL development to a toolbox. Without claim to be complete we refer to other toolboxes such as Stratego/SDF,²¹ MetaEdit+,²² and Meta Programming System (MPS).²³ An essential difference between all these toolboxes and DropsBox is the coverage of a DSL's life cycle. The mentioned toolboxes primarily focus on features for language implementation including concrete syntax specification, generation of editors, model validation, model transformation, model interpretation, and code generation. The DropsBox goes beyond these features by transferring further formalisms from language engineering and language application to modelling and metamodeling and by investigating phases of the DSL life cycle beyond design and implementation of language syntax and semantics.

Acknowledgments

This research is co-funded by the European Commission within the 6th and 7th Framework Programme projects MODELPLEX #034081, REWERSE #506779, and MOST #216691; the German Ministry of Education and Research (BMBF) within the projects feasiPLe and SuReal; the German Research Foundation (DFG) within the project HyperAdapt; the European Social Fund, Federal State of Saxony and SAP AG within the project #080949335 and the European Social Fund and Federal State of Saxony within the projects #080951806 (ZESSY), #80937064 and #080937065.

Also, we would like to thank all our students, the research community, and our industrial project partners for their valuable feedback on all the tools we have built over the last years. Without their constructive comments and their encouraging words, building DropsBox would have not been possible.

Last but not least we thank the anonymous reviewers for their constructive and helpful feedback. Their large effort in reviewing the paper helped us in enhancing the presentation of our work and to provide a more readable paper.

References

1. Greenfield, J., Short, K.: Software factories: Assembling Applications with Patterns, Models, Frameworks and Tools. Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (2003) 16–27

²⁰ <http://www.eclipse.org/workinggroups/oaw/>

²¹ <http://strategox.org/Sdf>

²² <http://www.metacase.com/MetaEdit.html>

²³ <http://www.jetbrains.com/mps/>

¹⁸ <http://www.eclipse.org/modeling/mdt/>

¹⁹ <http://www.eclipse.org/gmt/epsilon/>

2. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. *ACM Comput. Surv.* **37** (December 2005) 316–344
3. Fowler, M.: Language workbenches: The Killer-app for Domain Specific Languages. <http://www.martinfowler.com/articles/languageWorkbench.html> (2005)
4. Eclipse Foundation Eclipse Platform. <http://www.eclipse.org/> (April 2012)
5. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (January 2009)
6. Object Management Group: MOF 2.0 Core Specification. <http://www.omg.org/spec/MOF/2.0> (January 2006)
7. Efftinge, S., Voelter, M.: oAW xText: A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit. (2006)
8. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proc. of GPCE'06, New York, NY, USA, ACM (October 2006)
9. Krah, H., Rumpe, B., Völkel, S.: Efficient Editor Generation for Compositional DSLs in Eclipse. In: Proc. of DSM'07, Montreal, Quebec, Canada, Technical Report TR-38, Jyväskylä University, Finland (2007)
10. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: Proc. of the MoDELS 2006, Genova, Italy (October 2006)
11. Scheidgen, M. Textual Modelling Framework. <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/>
12. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In Cook, W.R., Clarke, S., Rinard, M.C., eds.: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, Reno/Tahoe, Nevada, ACM (2010) 444–463
13. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. *Electron. Notes Theor. Comput. Sci.* **203**(2) (2008) 103–116
14. Sloane, A.M.: Lightweight language processing in Kiama. In: Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III. GTTSE'09, Berlin, Heidelberg, Springer-Verlag (2011) 408–425
15. Nickel, U., Niere, J., Zündorf, A.: The FUJABA Environment. In: Proc. of 22nd International Conference on Software Engineering (ICSE'00), ACM (June 2000) 742–745
16. Geige, L., Buchmann, T., Dotor, A.: EMF Code Generation with Fujaba. In: Proc. of 5th International Fujaba Days, University of Kassel (October 2007)
17. Hesselund, A., Czarnecki, K., Wasowski, A.: Guided Development with Multiple Domain-Specific Languages. In: Model Driven Engineering Languages and Systems. Volume 4735 of LNCS. Springer Berlin / Heidelberg (2007) 46–60
18. Chimiak-Opoka, J., Demuth, B., Awenius, A., Chiorean, D., Gabel, S., Hamann, L., Willink, E.: OCL Tools Report based on the IDE4OCL Feature Model. In: Proceedings of the International Workshop on OCL and Textual Modelling Colocated with TOOLS Europe 2011, ICMT 2011, TAP 2011 and SC 2011. Volume 44., Electronic Communications of the EASST (2011)
19. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming* **69**(1-3) (2007) 27–34
20. Chimiak-Opoka, J.: OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In: Model Driven Engineering Languages and Systems. Volume 5795 of LNCS., Berlin / Heidelberg, Springer (2009) 665–669
21. Marconato, B. TOPCASED 2.3 tutorial Document generation (GenDoc). http://gforge.enseeiht.fr/docman/view.php/102/3325/TPC_2.3_GenDoc_tutorial.pdf
22. Eclipse Foundation Eclipse Intent. <http://wiki.eclipse.org/Intent> (April 2012)
23. Woodside, M., Petriu, D., Petriu, D., Shen, H., Israr, T., Merseguer, J.: Performance by unified model analysis (PUMA). In: Proceedings of the 5th international workshop on Software and performance, ACM (2005) 1–12
24. Petriu, D.B., Woodside, M.: An intermediate meta-model with scenarios and resources for generating performance models from UML designs. In: Software and Systems Modeling, Volume 6, Issue - 2. (2007) 163–184
25. Cortellessa, V., Di Marco, A., Inverardi, P.: Software performance model-driven architecture. In: Proceedings of the 2006 ACM symposium on Applied computing, ACM (2006) 1218–1223
26. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach For Automatic Model Composition. In: Workshops and Symposia at 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07). Volume 5002 of LNCS., Springer (June 2008) 7–15
27. Morin, B., Klein, J., Barais, O., Jézéquel, J.M.: A Generic Weaver for Supporting Product Lines. In: Proc. of 13th international workshop on Early Aspects (EA'08), ACM (May 2008) 11–18
28. Roychoudhury, S.: GenAWeave: A Generic Aspect Weaver Framework based on Model-Driven Program Transformation. PhD thesis, University of Alabama at Birmingham (September 2008)
29. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In: Transactions on Aspect-Oriented Software Development VI. Volume 5560 of LNCS., Springer (October 2009) 191–237
30. Kolovos, D., Paige, R., Rose, L., Polack, F.: Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology (JOT) – Special Issue for TOOLS Europe 2007* (2007)
31. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST* **3** (2006)

32. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: The Operation Recorder: Specifying Model Refactorings By-Example. In Arora, S., Leavens, G.T., eds.: *OOPSLA Companion*, ACM (2009) 791–792
33. Moha, N., Mahé, V., Barais, O., Jézéquel, J.M.: Generic Model Refactorings. In: *Model Driven Engineering Languages and Systems*. Volume 5795/2009 of LNCS. Springer (2009) 628–643
34. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *Proc. of 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*. (2005) 422–437
35. Kästner, C.: *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg (May 2010)
36. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: VML* – A Family of Languages for Variability Management in Software Product Lines. In van den Brand, M., Gray, J., eds.: *Proc. of the 2nd International Conference on Software Language Engineering (SLE'09)*, Revised Selected Papers. Volume 5969 of LNCS., Springer (March 2010) 82–102
37. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering. ASE '10*, New York, NY, USA, ACM (2010) 173–174
38. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. *Formal Methods for Open Object-Based Distributed Systems* (2006) 171–185
39. Triskell Project Team Kermeta: Triskell Metamodeling Kernel. <http://www.kermeta.org> (April 2012)
40. Matula, M. NetBeans Metadata Repository (2008)
41. Amelunxen, C., Klar, F., Königs, A., Röttschke, T., Schürr, A.: Metamodel-based Tool Integration with MOFLON. *30th International Conference on Software Engineering* (2008)
42. Dmitriev, S.: *Language Oriented Programming: The Next Programming Paradigm*. JetBrains onBoard 1(2) (2005)
43. Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a Fully Configurable Multi-user and Multi-tool CASE and CAME Environment. *Advanced Information Systems Engineering* (1996) 1–21
44. Corporation, M. The "Oslo" Modeling Language Specification (2009)
45. Cazzola, W., Speziale, I.: Sectional Domain Specific Languages. In: *Workshop on Domain-Specific Aspect Languages (DSAL'09)*, co-located with AOSD'09. (2009) 11–14
46. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program Analysis and Transformation in Java. *Rapport de recherche RR-5901*, INRIA (2006)
47. Klint, P., Lämmel, R., Verhoef, C.: Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14(3) (2005) 331–380
48. Antkiewicz, M., Czarnecki, K., Stephan, M.: Engineering of Framework-Specific Modeling Languages. *IEEE Transactions on Software Engineering* 35(6) (Nov.–Dec. 2009) 795–824
49. Visser, E.: WebDSL: A Case Study in Domain-Specific Language Engineering. In: *Generative and Transformational Techniques in Software Engineering II*. Volume 5235 of LNCS. Springer Berlin / Heidelberg (2008) 291–373
50. Herrmannsdoerfer, M., Ratiu, D., Wachsmuth, G. In: *Language Evolution in Practice : The History of GMF*. Volume 5969. Springer (2010) 3–22
51. Book, M., Gruhn, V.: A Dialog Flow Notation for Web-based Applications. In: *Proc. of 7th IASTED International Conference on Software Engineering and Applications (SEA 2003)*, ACTA Press (November 2003) 100–105
52. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming*. Volume 8 (3)., Elsevier (June 1987) 231–274
53. Seifert, M.: *Designing Round-Trip Systems by Change Propagation and Model Partitioning*. PhD thesis, Technische Universität Dresden (July 2011)
54. Kolovos, D.S.: *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York (June 2008)
55. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: *Proc. of 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*. Volume 5562 of LNCS., Springer (June 2009) 114–129
56. TU Dresden, Software Technology Group: EMFText Syntax Zoo. <http://www.emftext.org/zoo/> (April 2012)
57. Parr, T.J., Quong, R.W.: ANTLR: A Predicated-LL(k) Parser Generator. In: *Software – Practice and Experience*. Volume 25 (7)., John Wiley & Sons (July 1995) 789–810
58. Wirth, N.: What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? In: *Communications of the ACM*. Volume 20 (11)., ACM (1977) 822 – 823
59. Object Management Group: Human-Usable Textual Notation (HUTN) Specification, Version 1.0. <http://www.omg.org/spec/HUTN/1.0/> (August 2004)
60. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: *Proc. of ECMDA-FA*. Volume 5095 of LNCS., Springer (2008)
61. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference Attribute Grammars for Metamodel Semantics. In: *In Proc. of 3rd International Conference on Software Language Engineering (SLE '10)*. LNCS, Springer (October 2010)
62. Bürger, C., Karol, S.: Towards Attribute Grammars for Metamodel Semantics. Technical report, Technische Universität Dresden (March 2010)
63. Knuth, D.E.: Semantics of Context-Free Languages. In: *Theory of Computing Systems*. Volume 2 (2)., Springer (1968) 127–145
64. Knuth, D.E.: Semantics of Context-Free Languages: Correction. In: *Theory of Computing Systems*. Volume 5 (2)., Springer (1971) 95–96

65. Paakki, J.: Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.* **27**(2) (June 1995) 196–255
66. Ekman, T., Hedin, G.: The JastAdd System – Modular Extensible Compiler Construction. In: *Science of Computer Programming*. Volume 69 (1–3)., Elsevier (December 2007) 14–26
67. Hedin, G.: Reference Attributed Grammars. In: *Informatica (Slovenia)*. Volume 24 (3). (2000) 301–317
68. Boyland, J.T.: Remote Attribute Grammars. In: *Journal of the ACM*. Volume 52 (4)., ACM (July 2005) 627–687
69. Farrow, R.: Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-defined, Attribute Grammars. In: *Proc. of SIGPLAN symposium on Compiler construction (SIGPLAN 1986)*, ACM (July 1986) 85–98
70. Odersky, M., al.: An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland (2004)
71. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: *Model Driven Architecture – Foundations and Applications*. Volume 3748 of LNCS. Springer Berlin / Heidelberg (2005) 115–129
72. Hußmann, H., Demuth, B., Finger, F.: Modular Architecture for a Toolset Supporting OCL. In: *Proc. of 3rd International Conference on The Unified Modeling Language (UML 2000)*. Volume 1939 of LNCS., Springer (October 2000) 278–293
73. Bräuer, M., Demuth, B.: Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In: *Models in Software Engineering*. Volume 5002 of LNCS., Springer Berlin / Heidelberg (2008) 182–193 10.1007/978-3-540-69073-3_20.
74. Wilke, C., Thiele, M., Wende, C.: Extending Variability for OCL Interpretation. In: *Proc. of ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*. Volume 6394 of LNCS., Springer (October 2010) 361–375
75. Object Management Group: Object Constraint Language 2.2. <http://www.omg.org/spec/OCL/2.2/> (February 2010)
76. Meyer, B.: Applying "Design by Contract". *Computer* **25**(10) (1992) 40–51
77. Demuth, B., Hussmann, H., Loecher, S.: OCL as a Specification Language for Business Rules in Database Applications. In: *UML 2001 The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Volume 2185 of LNCS. Springer Berlin / Heidelberg (2001) 104–117 10.1007/3-540-45441-1_9.
78. Demuth, B., Wilke, C.: Model and Object Verification by Using Dresden OCL. In: *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice*, Ufa, Russia, July 25–31, 2009, Ufa, Bashkortostan, Russia, Ufa State Aviation Technical University (July 2009) 81
79. Bartho, A.: Creating and maintaining tutorials with DEFT. In: *Proc. of 17th IEEE International Conference on Program Comprehension (ICPC 2009)*, IEEE Computer Society (2009) 309–310
80. Wilke, C., Bartho, A., Schroeter, J., Karol, S., Aßmann, U.: Elucidative Development for Model-Based Specification. In: *TOOLS Europe 2012*. Volume 7304 of LNCS., Springer (2012) 321–336
81. Nørmark, K.: Requirements for an Elucidative Programming Environment. In: *Eight International Workshop on Program Comprehension*. (June 2000)
82. Knuth, D.E.: Literate Programming. In: *The Computer Journal*. Volume 27(2). (May 1984) 97–111
83. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: *Workshops and Symposia at ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems, Reports and Revised Selected Papers*. Volume 5002 of LNCS., Springer (2008)
84. Fritzsche, M., Gilani, W.: Model Transformation Chains to integrate Performance related Decision Support into BPM Tool Chains. In: *Invited submission for the post-Proc. of GTTSE 2009*, LNCS, Springer (2009)
85. XJ Technologies AnyLogic — multi-paradigm simulation software. <http://www.xjtek.com/anylogic/> (June 2009)
86. Eclipse Foundation ATLAS Transformation Language. <http://www.eclipse.org/m2m/at1> (April 2012)
87. Fritzsche, M., Johannes, J., Zschaler, S., Zhrebtsov, A., Terekhov, A.: Application of Tracing Techniques in Model-Driven Performance Engineering. In: *Proc. of 4th ECMDA Traceability Workshop (ECMDA-TW)*. (2008) 111–120
88. The AMW Project Team Atlas Model Weaver. <http://eclipse.org/gmt/amw/> (2012)
89. OMG – Object Management Group: UML profile for modeling and analysis of real-time and embedded systems. <http://www.omg.org/spec/MARTE/> (2007)
90. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. In: *Transactions on Aspect-Oriented Software Development VI*. Volume 5560 of LNCS., Springer (October 2009) 39–82
91. Henriksson, J.: A Lightweight Framework for Universal Fragment Composition—with an application in the Semantic Web. PhD thesis, Technische Universität Dresden (January 2009)
92. Johannes, J.: Component-Based Model-Driven Software Development. PhD thesis, Technische Universität Dresden (December 2010)
93. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*. Volume 1764 of LNCS., Springer (February 2000) 296–309
94. Johannes, J.: Developing a Model Composition Framework with Fujaba – An Experience Report. In: *Proc. of 7th International Fujaba Days*, TU Eindhoven (October 2009)
95. Johannes, J., Fernández, M.A.: Adding Abstraction and Reuse to a Network Modelling Tool using the Reuseware Composition Framework. In: *Proc. of 6th European Conference on Modelling Foundations and Applications (ECMFA'10)*. Volume 6138 of LNCS., Springer (2010)

96. Johannes, J., Aßmann, U.: Concern-based (de)composition of Model-Driven Software Development Processes. In: Proc. of ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). Volume 6395 of LNCS., Springer (October 2010)
97. Johannes, J.: Component-Based Model-Driven Software Development. PhD thesis, Technische Universität Dresden (January 2011)
98. Reimann, J., Seifert, M., Aßmann, U.: Role-Based Generic Model Refactoring. In: Proc. of ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10). (2010) 78–92
99. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring – Improving the Design of Existing Code. Addison Wesley (1999)
100. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G.: A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In: In Proc. of 2nd IFIP TC 2 Central and East European Conference on Software Engineering Techniques (CEE-SET'07), Revised Selected Papers. Volume 5082 of LNCS., Berlin, Heidelberg, Springer (2008) 252–266
101. Mens, T., Taentzer, G., Müller, D.: Challenges in Model Refactoring. In: Proc. 1st Workshop on Refactoring Tools, University of Berlin (2007)
102. Reenskaug, T., Per Wold, O.A.L.: Working with Objects: The OOram Software Engineering Method. Manning Publications, Greenwich, CT (1996)
103. Riehle, D., Gross, T.R.: Role Model Based Framework Design and Integration. In: Proc. of 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98). (1998) 117–133
104. Reimann, J., Seifert, M., Aßmann, U.: On the reuse and recommendation of model refactoring specifications. Software and Systems Modeling (2012) 1–18 10.1007/s10270-012-0243-2.
105. Frenzel, L. Eclipse Language Toolkit. <http://eclipse.org/articles/Article-LTK/ltk.html> (2006)
106. Mohamed, M., Romdhani, M., Ghedira, K.: Classification of Model Refactoring Approaches. In: Journal of Object Technology (JOT). Volume 8 (6)., ETH Zurich (September 2009) 143–158
107. Demuth, B., Chimiak-Opoka, J.: A Feature Model for an IDE4OCL. In: Proc. of Workshop on OCL and Textual Modelling (OCL2010). Volume 36., Electronic Communications of the EASST (2010)
108. Heidenreich, F., Kopcssek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion Proc. of 30th International Conference on Software Engineering (ICSE'08), New York, NY, USA, ACM (May 2008) 943–944
109. Heidenreich, F., Šavga, I., Wende, C.: On Controlled Visualisations in Software Product Line Engineering. In: Proc. of 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE'08), collocated with the 12th International Software Product Line Conference (SPLC'08). (September 2008)
110. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
111. Czarnecki, K., Eisenecker, U.W.: Generative Programming – Methods, Tools, and Applications. Addison-Wesley (June 2000)
112. OMG – Object Management Group: Unified Modeling Language (UML) Version 2.3. <http://www.omg.org/spec/UML/2.3> (May 2010)
113. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-0211990, Software Engineering Institute (1990)
114. Schmidt, M., Polowinski, J., Johannes, J., Fernandez, M.A.: An Integrated Facet-Based Library for Arbitrary Software Components. In: ECMFA 2010, LNCS 6138 Proc. Springer (2010) 261–276
115. Priss, U.: Faceted knowledge representation. Electronic Transactions on Artificial Intelligence 4 (2000) 21–33
116. Sacco, G.M., Tzitzikas, Y.: Dynamic Taxonomies and Faceted Search: Theory, Practice, and Experience. Volume 25 of The Information Retrieval Series. Springer (Aug 2009)
117. Polowinski, J.: Widgets for Faceted Browsing. M.J. Smith and G. Salvendy (Eds.): Human Interface 1 (2009) 601–610
118. University of California, Berkeley Flamenco. <http://flamenco.berkeley.edu/> (2010)
119. Massachusetts Institute of Technology (MIT) Exhibit. <http://simile.mit.edu/wiki/Exhibit> (2010)
120. Object Management Group CORBA project site. <http://www.corba.org> (2009)
121. OASIS UDDI community site. <http://uddi.xml.org/> (2009)
122. Wende, C., Thieme, N., Zschaler, S.: A Role-based Approach Towards Modular Language Engineering. 2nd International Conference on Software Language Engineering, (SLE 2009), Revised Selected Papers (2010)
123. Zschaler, S., Wende, C.: Collaborating Languages and Tools: A Study in Feasibility. Technical report, Technische Universität Dresden, Germany (July 2008)
124. Wende, C., Heidenreich, F.: A Model-based Product-Line for Scalable Ontology Languages. In Proc. of 1st International Workshop on Model-Driven Product Line Engineering (2009)
125. Seifert, M., Wende, C., Aßmann, U.: Anticipating Unanticipated Tool Interoperability using Role Models. Proc. of First Workshop on Model Driven Interoperability (MDI 2010), Oslo, Norway (2010)
126. Aßmann, U.: Invasive Software Composition. Springer (April 2003)
127. Andersen, E.P. Conceptual Modeling of Objects: A Role Modeling Approach. Ph.D. Thesis. Oslo, Norway, University of Oslo (1997)
128. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: Proc. of 2nd International Conference on Software Language Engineering (SLE'09). Volume 5969 of LNCS., Springer (March 2010) 374–383
129. Heidenreich, F., Johannes, J., Seifert, M., Wende, C., Böhme, M.: Generating Safe Template Languages. In: Proc. of 8th International Conference on Generative Programming and Component Engineering (GPCE'09), ACM (2009) 99–108

130. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Construct to Reconstruct – Reverse Engineering Java Code with JaMoPP. In: Proc. of International Workshop on Reverse Engineering Models from Software Artifacts (R.E.M. 2009). (October 2009)
131. Heidenreich, F., Johannes, J., Reimann, J., Seifert, M., Wende, C., Werner, C., Wilke, C., Aßmann, U.: Model-driven Modernisation of Java Programs with JaMoPP. In Fuhr, A., Hasselbring, W., Riediger, V., Bruntink, M., Kontogiannis, K., eds.: Joint Proceedings of the First International Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth International Workshop on System Quality and Maintainability (SQM 2011), March 1, 2011 in Oldenburg, Germany, CEUR Workshop Proceedings (March 2011) 8–11
132. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: JaMoPP: The Java Model Parser and Printer. Technical Report TUD-FI09-10, Technische Universität Dresden, Fakultät Informatik (September 2009)
133. Jakobsen, A.B., O’Duffy, M., Punter, T.: Towards a Maturity Model for Software Product Evaluations. In: Proc. of 10th European Conference on Software Cost Estimation (ESCOM 1999, Addison-Wesley (1999)
134. Shull, F., Singer, J., Sjøberg, D.I.K., eds.: Guide to Advanced Empirical Software Engineering. Springer (2007)
135. Zelkowitz, M.V., Wallace, D.R., Binkley, D.W.: Experimental validation of new software technology. Series on Software Engineering and Knowledge Engineering **12** (2003) 229–263