EuGENia Live: A Flexible Graphical Modelling Tool

Louis M. Rose
University of York
Deramore Lane, Heslington
York, YO10 5GH
United Kingdom
louis@cs.york.ac.uk

Dimitrios S. Kolovos
University of York
Deramore Lane, Heslington
York, YO10 5GH
United Kingdom
dkolovos@cs.york.ac.uk

Richard F. Paige
University of York
Deramore Lane, Heslington
York, YO10 5GH
United Kingdom
paige@cs.york.ac.uk

ABSTRACT

Designing a domain-specific language (DSL) is a collaborative, iterative and incremental process between domain experts and software engineers. Existing tools for **implementing** DSLs produce powerful and interoperable domain-specific editors, but are resistant to language change and require considerable technical expertise to use. We present EuGENia Live, a tool for **designing** (graphical) DSLs. EuGENia Live runs in a web browser, supports on-the-fly metamodel editing, and produces DSLs that can be exported and used with the Eclipse Modeling Framework. As well as presenting the design and implementation of EuGENia Live, we discuss potential benefits to our underlying approach, and challenges for future work on flexible modelling tools.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements

Keywords

Domain-specific languages, flexible modelling, model-driven engineering

1. INTRODUCTION

Domain-specific languages (DSLs) play an important role in model-driven engineering (MDE). In a recent survey of MDE practitioners, "almost 40%" of participants had used custom DSLs, 25% had used off-the-shelf DSLs, and only UML (used by 85% of participants) had been used more widely than DSLs [8]. Despite the significant adoption of custom DSLs for MDE, we believe that existing MDE tools for implementing DSLs and supporting tools are unsuitable for designing and developing a new DSL.

Like Fowler [5, pg.34] and Cho [1], we have observed that constructing a new DSL is a collaborative, iterative and incremental process. Domain experts and software engineers must work together to design the DSL. Changes to the DSL are frequent during the initial stages of DSL development. Implementing a DSL and supporting tools, however, requires considerable technical expertise (e.g., to construct a metamodel). Moreover, the process of constructing a DSL with MDE tools is somewhat resistant to change: metamodels and models are kept separate, models must always conform to their metamodel, and consequently managing model-metamodel co-evolution is a challenging task (see, for example, Cicchetti et al. [3], Herrmannsdörfer et al. [7], or our earlier work [11]).

Flexible modelling tools (e.g., [1, 4, 6, 12]) seek to combine free-form modelling (e.g., sketching on a whiteboard) and more formal modelling (e.g., modelling with a MDE tool). Flexible modelling tools sacrifice some formality to facilitate experimental and exploratory modelling of the domain, but consequently cannot provide the powerful domain-specific editors generated by MDE tools. We believe that flexible modelling is particularly valuable for the design of DSLs, but as the design of a DSL solidifies there is increasing value in implementing the DSL with a MDE tool.

In this paper, we introduce EuGENia Live, a flexible modelling tool that aims to reduce the effort required to construct DSLs, and that exports models and metamodels for continuing the implementation of a DSL with the MDE tools available in the Eclipse Modeling Project. The remainder of this paper is structured as follows. Section 2 reviews existing work on modelling tools, categorising them into *rigid* and *flexible* tools. Section 3 introduces EuGENia Live, a flexible modelling tool that we have developed which seeks to facilitate collaborative, iterative and incremental development of DSLs. Using an example, section 4 demonstrates the key features of EuGENia Live. Section 5 considers the benefits, drawbacks and challenges of the modelling approach underpinning EuGENia Live, and section 6 concludes and explores possible directions for future work.

2. RIGID AND FLEXIBLE MODELLING

We now review and compare rigid and flexible modelling tools, with a focus on their use for specifying graphical DSLs. *Rigid modelling tools* – which include MDE modelling tools and language workbenches – require that models conform to a language specification (e.g., a metamodel or a grammar), and constructing a domain-specific model with a rigid modelling tool involves first constructing a language specification for the domain. *Flexible modelling tools* seek to combine free-form modelling with rigid modelling, and normally allow domain-specific models to be initially constructed using a general-purpose modelling language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2012 ACM 978-1-4503-1804-4/12/10 ...$15.00.

2.1 Rigid Modelling Tools

MDE and language-oriented programming are two communities in which rigid modelling tools are prevalent. In both communities, domain-specific programs conform to a DSL specification, which is used to generate DSL editors and other domain-specific tooling (e.g., by specifying model transformations and code generators).

2.1.1 MDE Modelling Tools

In the MDE community, a DSL is often defined using models, transformations and other model management operations. The Eclipse Modeling Project, arguably the most widely-used MDE environment today, provides the Eclipse Modeling Framework (EMF) and several related tools for specifying DSLs. EMF is used to specify abstract syntax.

Graphical concrete syntax for a language defined with an EMF model can be specified using frameworks such as Graphiti[1] and the Graphical Modeling Framework (GMF). Users of GMF create four interrelated models that specify different parts of the graphical concrete syntax, and GMF from the four models generates Java code for the corresponding graphical editor. Graphiti is a Java API for creating graphical tools. Users of Graphiti write Java code that uses the API to specify a graphical editor that stores data according to the structure specified by an EMF model. GMF and Graphiti are large and powerful frameworks, and consequently their use requires considerable technical expertise (as evidenced for GMF by Wienands and Golm [14]). EuGENia [9] and Spray[2] seek to reduce the amount of technical expertise required to specify a DSL with GMF and Graphiti respectively, by providing higher-level languages that map onto a subset of the underlying framework.

2.1.2 Language-Oriented Tools

Language-oriented programming [13] involves constructing, extending, combining and using different languages together to construct a system. *Language workbenches* are programming environments that seek to address the requirements of language-oriented programming. Although more than a dozen language workbenches were presented at the 2011 and 2012 Language Workbench Competitions[3], there are very few publications on these tools.

There are relatively few graphical language workbenches. MetaEdit+ and Obeo Designer are commercial graphical language workbench tools. MetaEdit+ provides mechanisms for specifying a language by defining abstract syntax, visual concrete syntax and semantics (via code generators). MetaEdit+ uses a proprietary persistence format and hence models and languages constructed with MetaEdit+ cannot be used with other modelling tools or language workbenches. Obeo Designer is a graphical language workbench tool that combines many MDE tools. Visual concrete syntax is specified using GMF, which was described in Section 2.1.1.

2.2 Flexible Modelling Tools

Flexible modelling tools seek to combine some of the characteristics of free-form modelling (e.g., sketching using a whiteboard or an office tool) with some of the characteristics of rigid modelling tools (e.g., structured data). A range of approaches to flexible modelling are being explored, and initial work has been published at, for example, Flexible Modeling Tools workshops[4]. Unlike rigorous modelling tools, flexible modelling tools do not necessarily require a metamodel to be specified prior to the construction of models.

Cho [1] and Desmond et al. [4] describe approaches that infer a metamodel after models have been constructed. Inference approaches require a set of example models, which are constructed by the user with a generic concrete syntax (comprising, for example, shapes and lines). The modelling tool then infers language-specific syntax from the example models, and the language-specific syntax can be used in future modelling sessions. Inferring a metamodel from example models is challenging, and it is not yet clear whether inference can be fully automated: Cho [1] explores a fully automated inference process that uses metamodel design patterns to improve the structure of inferred languages; while Desmond et al. [4] explores a semi-automated inference process that is guided by the user.

As an alternative to inferring a metamodel, Gabrysiak et al. [6] proposes the co-design of models and metamodels via incremental refinement of an initial metamodel. The initial metamodel is based on a well-understood domain (e.g., Petri nets), and is changed to better suit the domain that is being modelled.

2.3 Flexible versus Rigid Modelling Tools

Rigid and flexible modelling tools are now compared with respect to three distinguishing characteristics.

2.3.1 Model Representation

In rigid modelling tools, all models are structured: they conform to a metamodel. The environments in which rigid modelling tools are integrated exploit structured models to provide other functionality (e.g., language-specific model editing capabilities). MDE environments are particularly reliant on structured models, as model transformations are typical specified with respect to a metamodel (e.g. "transform all of the instances of the *Net* metaclass"). By contrast, flexible modelling tools focus more on designing the appearance (concrete syntax) than the structure (abstract syntax) of a DSL, and consequently the languages designed with flexible modelling tools might not be amenable to model transformation. This drawback is discussed further in section 5.

2.3.2 Technical Expertise

Rigid modelling tools assume technical expertise, and hence are normally used by software engineers. For example, metamodelling languages might assume understanding of object-orientation (for specifying abstract syntax) or BNF for (specifying textual concrete syntax). Flexible modelling tools seek to reduce the technical expertise required to specify a modelling language via automation or refinement.

2.3.3 Change Management

Designing a DSL in collaboration with domain experts typically involves discussing example models rather than by discussing metamodels [10]. Similarly, designing or modifying a DSL with a flexible modelling tool starts by changing example models, and a metamodel is constructed or inferred from the example models. By contrast, modifying a DSL

[1] http://www.eclipse.org/graphiti/
[2] http://code.google.com/a/eclipselabs.org/p/spray/
[3] http://www.languageworkbenches.net
[4] http://www.ics.uci.edu/~nlopezgi/flexitoolsICSE2011/

Goal	Requirement	Approach taken by EuGENia Live
1	Allow DSL specifications to be exported to a platform for implementing DSLs.	Option to export a user-defined DSL as an Ecore model for use with EMF, and GMF via EuGENia [9].
2	Minimise the time and technical expertise required for installation.	Implemented as a web application, and runs in any modern web browser.
2	Provide structures that reduce the time required to start developing a new DSL.	Built-in metamodels (for Petri nets and state machines) that can be re-used and customised when designing a new DSL.
2	Support both MDE experts and domain experts in specifying a DSL.	Users can choose between two metamodelling notations: JSON (a human-readable form of XML) and EuGENia [9].
3	Allow users to quickly switch between editing and testing a DSL.	A drawing editor whose palette (i.e. metamodel) can be changed on-the-fly by the user.

Table 1: Requirements for and approaches taken in EuGENia Live.

with a rigid modelling tool typically involves first changing the metamodel, possibly re-generating code for the supporting tools for the DSL, and then testing the modified DSL. Consequently, it is arguably easier to change DSLs implemented with flexible modelling tools than to change DSLs implemented with rigid modelling tools.

3. EuGENia Live

This section presents the design and implementation of EuGENia Live, a graphical modelling tool that seeks to combine the benefits of flexible and rigid modelling tools with the aim of facilitating collaborative, iterative and incremental specification of DSLs. To this end, EuGENia Live has the following goals:

1. **Interoperate with existing model management tools** (inspired by rigid modelling tools, such as EMF and GMF).
2. **Reduce the technical expertise required to specify a DSL** (inspired by inference-based flexible modelling tools [1, 4]).
3. **Minimise the time taken to change a DSL** (inspired by flexible modelling tools).

Table 2.3.2 identifies the way in which the above goals have been translated into user requirements, and summarises the approach taken in EuGENia Live to address each requirement. The remainder of this section describes the approaches in more detail. First, the user interface of EuGENia Live is summarised, and then the domain model – which is key to addressing many of the requirements – is discussed. An alpha version of EuGENia Live is available at `http://eugenialive.herokuapp.com`

3.1 User Interface

The user interface of EuGENia Live comprises three views, and allows DSLs to be designed by creating drawings (models) and palettes (metamodels). Each view is now discussed.

The *DrawingNavigator* (not shown) is the primary view, and allows drawings to be created, deleted and opened. When creating a new drawing, the user can select either an empty or a built-in palette.

The *DrawingEditor*, figure 1(a), provides a canvas, toolbox and property sheet editor that allows users to test a palette by creating a drawing that uses the palette. The toolbox on the left hand-side of the *DrawingEditor* allows the user to construct a drawing. New drawing elements are created by first selecting an item in the toolbox, and then clicking on the canvas (on the right-hand side of the *DrawingEditor*).

The *PaletteEditor*, figure 1(b), is accessed by double-clicking on the items in the toolbox of the *DrawingEditor*, and allows node types and edge types to be modified. To modify a node type or edge type, the user changes a serialisation of the node type or an edge type using either JSON (an XML-like notation) or EuGENia [9] concrete syntax. The abstract syntax for the node types and edge types of EuGENia Live is now discussed.

3.2 Domain Model

The domain model (figure 2) comprises the concepts necessary to create and use a graphical DSL. In EuGENia Live, a model is an instance of *Drawing* and can contain any number of *Nodes* and *Edges*. Each instance of *Node* and *Edge* can contain a number of *Slots* which are used to store property values, such as the name of the *Node* or *Edge*.

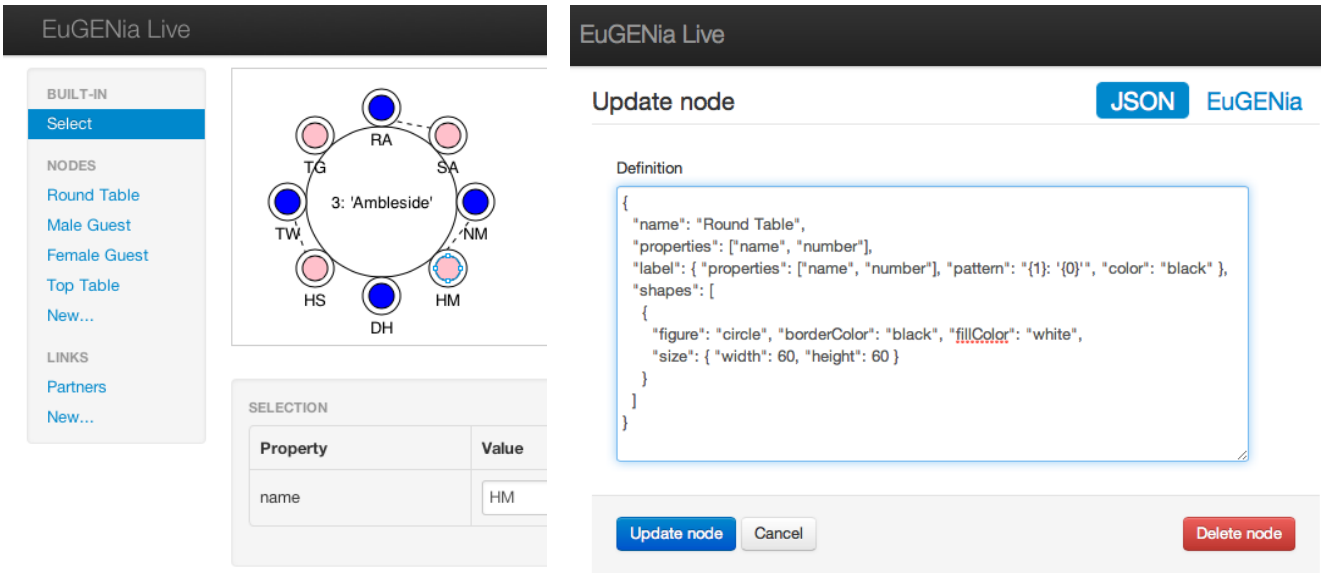
Every instance of *Drawing* references a *Palette*, which defines the metamodel for that *Drawing*. A *Palette* comprises *NodeTypes* and *EdgeTypes*. Each *NodeType* and *EdgeType* has a name and contain any number of *Properties*. Each *NodeType* and *EdgeType* may optionally specify a *Label*, which provides one piece of concrete syntax for a *Node*. The remaining concrete syntax is specified using one or more *Shapes* for a *NodeType* and a *Line* for *EdgeTypes*. In future work, we anticipate supporting additional syntax (e.g., arrowheads for *Lines*).

Two key characteristics of the domain model are that model and metamodel elements are specified together and that the relationship between model and metamodel elements are realised using an association rather than with instantiation (e.g., see the relationships between *Edge* and *EdgeType* in figure 2). Consequently, a palette can be changed on-the-fly, whilst a drawing is created.

3.3 Implementation Notes

We have implemented EuGENia Live using HTML5 and Coffeescript[5]. *Drawings* are rendered using the HTML5 canvas element and the Paper.js[6] vector graphics framework. The domain model and UI are implemented with Spine[7] and Twitter Bootstrap[8], respectively.

[5]A language that is similar to Ruby and that compiles to cross-browser Javascript, `http://coffeescript.org`
[6]`http://paperjs.org`
[7]`http://spinejs.com/`
[8]`http://twitter.github.com/bootstrap/`



(a) The *DrawingEditor* view.

(b) The *PaletteEditor* view.

Figure 1: Using the views of EuGENia Live to define a seating plan diagram and its language definition.

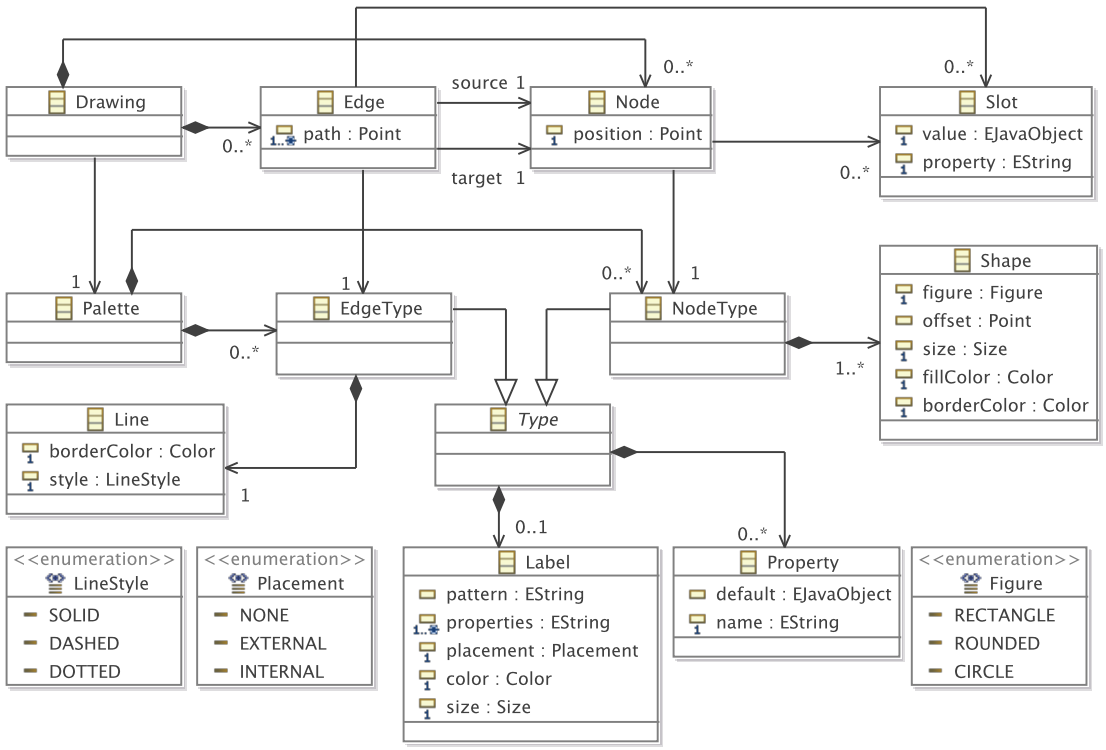


Figure 2: UML class diagram of the domain model of EuGENia Live.

4. EXAMPLE

We now consider an example of designing a graphical DSL. The purpose of the example is to contextualise a discussion of the preliminary version of EuGENia Live; further development and application to many more examples will be necessary for a rigorous evaluation. The DSL we now consider is used for specifying seating plans. The seating plan DSL captures concepts such as tables and guests. The purpose of the seating plan DSL is to create allocations of guests to tables, and then to check properties of the allocation such as "every table must have male and female guests" and "partners should be seated on the same table". The seating plan DSL was developed collaboratively by a technical and a non-technical stakeholder. The remainder of this section discusses how EuGENia Live was used during the three iterations of the design of the seating plan DSL (figure 3).

4.1 Iteration (a): Re-using State Machines

The first iteration of the seating plan DSL focused on supporting two key concepts: *tables* and *guests*. Development of the language began by creating a new drawing. To bootstrap development of the seating plan DSL, the built-in state machine palette was chosen for the new drawing. Initially, some nodes from the state machine palette were created (by clicking on their name in the toolbox, and then clicking on the canvas). One of the *NodeTypes* from the built-in state machine palette, *state*, was then customised to to make it larger and to change its name to *table*. Customisation of a *NodeType* is achieved by by double-clicking on its name in the palette, and then updating its definition using the palette editor, as shown in figure 1(b). Similar customisation was carried out to adapt the *start state NodeType* to form a *guest NodeType*. The new *NodeTypes* were tested by constructing the drawing in figure 3(a).

4.2 Iteration (b): Adding Labels & Genders

The second iteration involved adding labels to the table and guest *NodeTypes*. Using the palette editor, *number* and *name* properties were added to the *table NodeType*, along with a label that displays the values of the *number* and *name* properties. Also in this iteration, conversation between the stakeholders revealed that it was important to model the gender of guests in order to specify an intra-model constraint: each table must seat male and female guests. To this end, the original *guest* type was renamed to *female guest* and copied to produce a *male guest* type. This change is discussed further in section 5. After making these changes and adding numbers and names to the nodes, the drawing used to test the DSL was as shown in figure 3(b).

4.3 Iteration (c): Adding Seats & Partners

The final iteration involved adding two further concepts to support the definition of additional intra-model consistency constraints. Firstly, the *partners EdgeType* was added for representing that two guests are attending the event together (and hence must be seated on the same table). New *EdgeTypes* or *NodeTypes* are created by clicking on the "new" hyperlink in the drawing editor, as shown in figure 1(a).

Secondly, the *table* type was extended to include a containment relationship for *guests* (and hence tables can be checked to ensure that they are full). This could have been implemented by adding a *seat* type. Instead, the *table* type was changed by adding an extra (circle) shape for each seat at the table. Note that this extension exploits the one-to-many relationship between *NodeTypes* and *Shapes* in figure 2. This change to the DSL identified a new requirement for EuGENia Live that relates to native support for containment relationships, which we plan to investigate soon.

After creating new edges with the *partners EdgeType* and aligning the guest nodes with the "seats" (small circles) in the table node, the drawing used to test the DSL was as shown in figure 3(c).

5. DISCUSSION

Through our preliminary work on EuGENia Live we have identified and now discuss some potential benefits of and challenges for combining rigid and flexible modelling.

5.1 Benefit: Incremental Model Migration

Model migration is the process of re-establishing the conformance of model following changes to its metamodel. Existing approaches to model migration [3, 7, 11] assume that several metamodel changes occur together before model migration. In EuGENia Live, a model and its metamodel are designed together, and model migration can be incremental: conformance problems can be reported and potentially resolved as the metamodel is changed. For example, in section 4) the guest template was split into a male guest and a female guest template. Model migration was immediately realised by deleting and re-creating some of the existing guest nodes. To assist in incremental model migration, EuGENia Live could provide migration tools, such as a mechanism for changing the type of existing nodes.

5.2 Benefit: Multiple Syntaxes

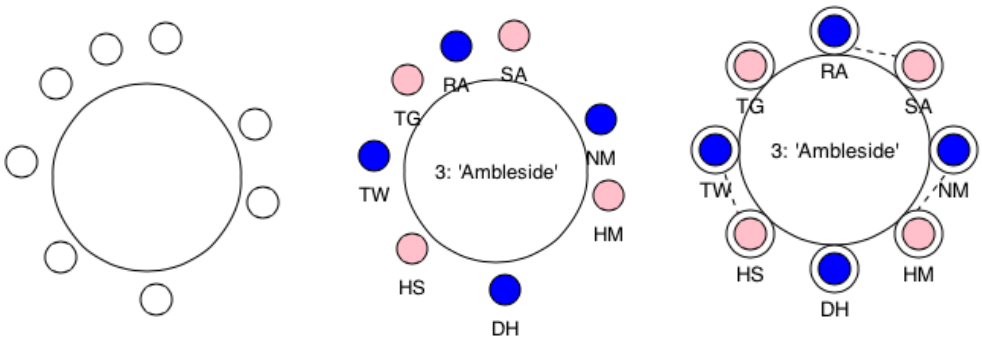
The concrete syntax of a model might need to vary for different stakeholders or different use cases. For example, the seating plan DSL is used to generate a tabular seating plan, which is shared with a further set of stakeholders (event organisers). The tabular seating plan is currently created using a model-to-text transformation. Instead, EuGENia Live might support multiple concrete syntaxes for each diagram. To support multiple concrete syntaxes, the domain model (figure 2) would be changed to allow each *Drawing* to have more than one *Palette*, and the view layer would need to provide a means for switching between concrete syntaxes.

5.3 Challenge: Language Quality

Although we have only produced a few small DSLs with EuGENia Live to date, the abstract syntax always contains some duplicated properties, probably because is no inheritance relationship between templates. Due to the way in which templates are presented to the user in EuGENia Live, it is not yet clear how we might represent language concepts that have no concrete syntax and exist only to reduce duplicated from the abstract syntax. Cho and Gray [2] propose applying metamodel design patterns to metamodels to reduce duplication and to fix other metamodel design issues, and a similar approach might be applied to the metamodels produced by EuGENia Live before they are exported.

5.4 Challenge: When to Model Rigidly

Rigid modelling tools can produce DSL editors that have functionality comparable to editors for general-purpose languages, such as code completion and syntax highlighting (for textual languages) and automated layout algorithms (for



(a) Initial implementation. (b) Adding labels & genders. (c) Adding seats & partners.

Figure 3: The iterative and incremental development of the seating plan DSL with EuGENia Live.

graphical languages). Similar functionality might be difficult to produce with flexible modelling tools due to the lack of an explicit language specification. EuGENia Live focuses on rapid prototyping of a DSL, and provides the ability to export diagrams and palettes to GMF (via its EuGENia notation). A key question for future research will therefore be to indicate at what point it is most cost effective to switch from a rapid prototyping tool, such as EuGENia Live, to a more powerful tool, such as GMF.

5.5 Challenge: Closer Collaboration

Designing a DSL is a collaborative process between domain experts and software engineers. In the current version of EuGENia Live, we have sought to increase collaboration between domain experts and software engineers by aiming to reduce the time between language iterations and by reducing the technical expertise required to specify DSLs. We believe that, as a consequence, domain experts will be more involved in the design process. It is not yet clear whether this is the case. We believe that additional features will be needed to encourage closer collaboration, such as support for real-time collaboration (à la Google Docs).

6. CONCLUSIONS

Rigid and flexible modelling tools provide different benefits. By seeking to combine these benefits of rigid and flexible modelling, EuGENia Live aims to facilitate collaborative, incremental and iterative design of graphical DSLs. EuGENia Live encourages the co-design of models and metamodels to support iterative and incremental development, and focuses on reducing the technical expertise required to specify DSLs in order to encourage collaboration. EuGENia Live does not provide the advanced language-specific features of DSL editors generated with rigid modelling tools, but instead bootstraps implementation of DSLs by exporting metamodels for use with EMF and GMF.

In future work, we will explore the feasibility of our proposed modelling approach by releasing EuGENia Live as part of the Epsilon[9] project. In the short-term, we anticipate extending EuGENia Live to support type and cardinality constraints for properties and edges and additional concrete syntax for containment relationships. In the long-term, we plan to contribute flexible modelling tools to Eclipse.

[9]`http://www.eclipse.org/epsilon`

7. REFERENCES

[1] H. Cho. A demonstration-based approach for designing domain-specific modeling languages. In *OOPSLA Companion*, pages 51–54. ACM, 2011.

[2] H. Cho and J. Gray. Design patterns for metamodels. In *Proc. SPLASH Workshops*, pages 25–32. ACM, 2011.

[3] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *Proc. EDOC*, pages 222–231. IEEE Computer Society, 2008.

[4] M. Desmond, H. Ossher, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, and S. Krasikov. Towards smart office tools. In *Proc. FlexiTools Workshop*, 2010.

[5] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[6] G. Gabrysiak, H. Giese, A. Luders, and A. Seibel. How can metamodels be used flexibly? In *Proc. FlexiTools Workshop*, 2011.

[7] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE - automating coupled evolution of metamodels and models. In *Proc. ECOOP*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[8] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proc. ICSE*, pages 471–480. ACM, 2011.

[9] D. Kolovos, L. Rose, S. Abid, R. Paige, F. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *MoDELS (1)*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.

[10] M. Kuhrmann. User assistance during domain-specific language design. In *Proc. FlexiTools Workshop*, 2011.

[11] L. Rose, D. Kolovos, R. Paige, and F. Polack. Model migration with Epsilon Flock. In *Proc. ICMT*, volume 6142 of *LNCS*, pages 184–198. Springer, 2010.

[12] B. Volz, M. Zeising, and S. Jablonski. The Open Meta Modeling Environment. In *Proc. FlexiTools Workshop*, 2011.

[13] M. Ward. Language-oriented programming. *Software — Concepts and Tools*, 15(4):147–161, 1994.

[14] C. Wienands and M. Golm. Anatomy of a visual domain-specific language project in an industrial context. In *Proc. MoDELS*, volume 5795 of *LNCS*, pages 453–467. Springer, 2009.