Journal of Universal Computer Science, vol. 26, no. 9 (2020), 1148-1176
submitted: 14/5/20, accepted: 25/9/20, appeared: 28/9/20

Model Driven Software Engineering Meta-Workbenches: An XTools Approach

Tony Clark
(Aston University, Birmingham, United Kingdom
tony.clark@aston.ac.uk)

Jens Gulden
(Utrecht University, Utrecht, The Netherlands
jensgulden@acm.org)

Abstract: Model Driven Software Engineering aims to provide a quality assured process for designing and generating software. Modelling frameworks that offer technologies for domain specific language and associated tool construction are called language workbenches. Since modelling is itself a domain, there are benefits to applying a workbench-based approach to the construction of modelling languages and tools. Such a framework is a meta-modelling tool and those that can generate themselves are *reflective* meta-tools. This article reviews the current state of the art for modelling tools and proposes a set of reflective meta-modelling tool requirements. The XTools framework has been designed as a reflective meta-tool and is used as a benchmark.
Key Words: Model Driven Engineering, Meta Modelling, Reflexive Software Tools.
Category: D 2, D 2.2

1 Model Based Tool Interfaces

The use of models in system development is widespread and its systematic use is termed Model Driven Software Engineering (MDSE, [Beydeda et al., 2005, Stahl and Voelter, 2006]). Tools that support aspects of the MDSE process often support standards such as the Unified Modeling Language (UML, [OMG, 2015]). Other tools support the creation and use of domain specific modelling languages.

Although tools for MDSE have a common aim there has been little scientific attention to the subject of applying MDSE to itself. Our claim is that the application of MDSE to the construction of MDSE tooling is a valid field of study that will allow us to build reflexive software engineering product line tooling, adaptive tooling, self-healing tooling, and tooling that is capable of self improvement by generating new generations. The starting point for such a vision for reflexive MDSE tooling is to identify a language that is capable of describing itself. It is likely that there will be many such languages, however if we can demonstrate that it is possible, variations can be generated by following the process. Tool meta-circularity aims to achieve a single system that is capable of generating a family of systems including itself that are characterised by a set of common capabilities. The benefits of achieving meta-circularity include:

improved **quality** control, a reduction in the **effort** required and an increase in **tool compatibility**.

This article describes an approach to reflexive MDSE tooling called XTools which is a language that has been designed with a representative set of MDSE tool features. It is able to reflect on its own behaviour and to generate tools including itself *because* it is built on a meta-circular kernel language called XCore. We provide an overview of XCore and identify the key features that lead to XTools being able to generate a family of modelling tools including itself.

Our method is to review the MDSE process from the perspective of tooling and identify common tool requirements leading to a set of concepts that form our domain choice for XTools. We then describe how these concepts are implemented in terms of XCore. We demonstrate the novelty of our approach by reviewing other MDSE tooling, identifying their key MDSE features, and analyzing to what extent the particular tool can be used to build itself. Our claim is that XTools is the only tool that can be used to build itself. The contribution of this work is as follows: (1) We identify a common representative set of requirements for DSL tooling. Such tools are typically object-oriented and offer encapsulation and extensibility in the usual way. Our proposition is that an MDSE workbench that is *meta-circular* and extensible captures the class of tools based on the requirements as a single system. (2) We describe a meta-circular extensible workbench called XTools. (3) We show how XTools is constructed through a simple meta-circular language called XCore that is self-descriptive in terms of both structure and behaviour. (4) We contribute a review of established MDSE workbenches with respect to the criteria for meta-circularity and extensibility concluding with a comparison table showing the extent to which the common features are supported by each candidate workbench.

Section 2 provides a domain analysis of MDSE language workbenches and produces a set of requirements which are then shown to be supported in a reflexive way by the XTools package within the XModeler toolkit in section 3. Established MDSE technologies are reviewed in section 4 leading to a comparison with respect to meta-circular MDSE capabilities in section 5. The comparison is not intended to be a competitive evaluation since it is not complete with respect to MDSE tooling being limited to *meta-circularity.* Although we conclude that XTools is more mature with respect to meta-circularity, there are other features, *e.g.* engineering maturity, where workbenches such as Sirius [Viyović et al., 2014] and MetaEdit+ [Tolvanen and Kelly, 2009] win out.

2 Domain Analysis

MDSE workbenches support the construction of MDSE tools where the modelling domain contains language and tool definition concepts. A human user must use languages to express concepts which are typically supported in the following ways:

Diagrams: It should be possible to define a language in terms of graphical display elements. Diagrams should support a range of user generated events and respond accordingly and it should be possible to express constraints on diagrams to ensure that the syntax is well-formed.

Trees: Language elements can often be grouped into categories and can be arranged in terms of a parent-child relationship. A tree browser provides a convenient way of organising model elements written in a particular language and of providing access to domain-specific functionality over the elements.

Forms: Model elements often have properties whose values can be set by a user. The properties of an element can be expressed using a form that lists the names of the properties and allows the values to be set.

Text: It is often convenient to support concept construction through graphical elements that show relationships whose visual features convey semantics. When model features become large and complex it is necessary to be able to mix graphical and textual representations.

Events: When creating models a user will typically interact with a user interface through a keyboard or mouse. The interactions give rise to a fixed range of events that cause changes to the model under construction.

The basic concepts described above capture the main features of MDSE tooling and are therefore the basis of any meta-circular definition of an MDSE workbench. As noted in [Paige et al., 2017] "Despite the inroads that MD[S]E has made in industry, a recurring complaint and obstacle for industrial organisations considering MD[S]E is the lack of sufficient tool support". The article goes on to propose a challenge for MDSE tooling that achieves "a uniform, cohesive, and seamless integrated experience when progressing from concept to deployed system". In order to address this issue it is important to understand the key features required by MDSE tooling and how they can be supported to achieve the desirable characteristics.

A survey of MDSE [da Silva, 2015] reviews model-driven engineering in terms of questions such as the definition of a model, its relationship to meta-models, and the key facets of modelling languages. This survey finds that diagrams are a key feature of virtually all modelling languages and refers to this as being an appropriate mechanism to express the *structural semantics* required by MDSE. However, the authors go on to make the point that graphical models do not scale as well as text or tables (forms) where these approaches may be more appropriate [Voelter et al., 2013]. In addition, due to a variety of syntax it should be possible to combine both.

A DSL requirement for MDSE has led to the idea of a *language workbench* [Erdweg et al., 2013] that promotes reuse through a collection of tools for engineering and subsequently using languages. An example of such a workbench is the Graphical Modelling Framework (GMF) [Eclipse Foundation, b, Gronback, 2009] which provides a collection of meta-languages (Ecore, OCL, a tooling model, and a graph diagram definition language) for defining different aspects of DSLs including its abstract syntax, its behaviour, and its concrete syntax (Sect. 4.1). A danger with this approach is that the different meta-languages need to integrate cleanly and should be extensible both individually and in combination, therefore we would like to aim for closely integrated languages for the different facets.

DSL design guidelines [Karsai et al., 2014] include a requirement to use descriptive notations and to make elements distinguishable. This implies a degree of control over the interface features (diagram elements, icons, etc.) that are provided to the user of the DSL. Given that languages may be used by different stakeholders, it also implies that it may be necessary to tailor the notations both in terms of a-priori configuration and at run-time.

The design guidelines also advocate the provision of organisational structures for models that reflect the domain and the users' expectations. Our proposal is that these structures can be provided through support for domain-specific tree-based tools. Understanding notations involves principles of semiotics and fundamental Gestalt theory [Gulden, 2016] leading to criteria such as Perceptual Discriminability, Semiotic Clarity, and Cognitive Fit [Moody, 2009]. Requirements for cognitive perception are also discussed in [Gulden and Reijers, 2015, Gulden et al., 2016].

Nesting occurs in a number of standard modelling languages including state-machines, packages, and business processes. As noted in [Karsai et al., 2003], hierarchical nesting is an abstract structuring principle and therefore should be supported by meta-tools for DSL development.

The domain analysis above leads us to a collection of core requirements for DSL meta-tools that are defined in Table 1. The requirements cover both notation features such as the ability to represent language concepts using trees and diagrams, language organisation features such as nesting, and tool facilities such as the ability to access the semantics of a model written in the language.

3 The XTools Approach

XModeler is a tool for language and tool engineering [Clark et al., 2015b, Clark et al., 2015a, Clark and Willans, 2013] that is based on a single meta-circular meta-model called XCore that is comparable to ECore [Gronback, 2009, Steinberg et al., 2009]. Everything in XModeler is an object, including classes and meta-classes, which facilitates the construction of reusable languages and tools. XModeler is implemented as a small VM written in Java; XCore is created

RQ-1 **Diagrams:** A key feature of most DSLs is the use of diagrams. Therefore a language for defining DSL tools must support the definition and manipulation of diagrams.

RQ-2 **Forms:** Model elements have properties that must be set as part of the modelling activity. Forms can be used to display and edit properties and their values.

RQ-3 **Text:** There are occasions that it is more convenient to use a text-based approach to modelling. Therefore, a basic requirement is to provide languages for features including, *e.g.*, actions and constraints. Since DSLs may be text-based it should be possible to define domain-specific textual language features.

RQ-4 **Trees:** Key structuring aspects of tooling, such as ownership, are conveniently represented using trees.

RQ-5 **Abstraction:** A meta-tool for DSL construction should provide suitable abstractions that shield the language engineer from implementation concerns. For example, the details of handling events and the management of graphics should be hidden behind suitable meta-language constructs. It should be possible to access implementation detail where necessary.

RQ-6 **Notations:** A key feature of DSL tools is the compatibility between the notations provided and the domain concepts. Therefore a meta-tool should provide appropriate features for defining notation and symbols including configuration at run-time.

RQ-7 **Semantics:** A DSL-based tool should not be limited to an editor: it should have semantics. The semantics may be expressed in terms of translation to another format including source code, in terms of constraints that are checked during model construction, or direct execution. A meta-technology should support all of these.

RQ-8 **Integration:** All aspects of a DSL tool including diagrams, forms, text, trees, and semantics, should be fully integrated. This implies that the use of different third-party meta-technologies to configure aspects of a tool are likely to be limiting.

RQ-9 **Nesting:** Recursion is a key abstraction mechanism that should be supported by meta-tool definition. If a DSL is naturally recursive then the tooling should be fully aware and support it through nesting that can be provided in a number of ways. Trees and text-based languages support nesting in a straightforward way. Diagrams can support nesting in several different ways, for example directly on a diagram, or through separately selected sub-diagrams.

RQ-10 **Patterns:** Abstraction is supported through the definition of patterns. Since any non-trivial DSL tool is likely to be complex, it is important that the meta-tooling supports abstraction through parametric patterns so that definitions can be composed from separately verified components.

RQ-11 **Extension:** The meta-language provided to define DSLs should support extension so that tools can reuse basic definitions.

Figure 1: Requirements for MDSE DSL Tooling

via a small bootstrap file and then the XModeler toolset is bootstrapped from the basic XCore. An executable language called XOCL, based on standard OCL, is implemented in XCore as a compiler and interpreter.

The language engineering features of XModeler are used to create a meta-circular language called XTools that provides declarative mechanisms for diagrams, trees and forms. XTools languages are interpreted at run-time within the XModeler environment and therefore fully integrate with all other XModeler features.

This section describes the features of XTools and how the XTools architecture achieves meta-circularity. Section 3.1 describes XCore the meta-circular language that is the basis for XTools in XModeler. Section 3.2 provides an overview of the XTools language as implemented in XCore and the general purpose event mechanism that is used to support a Model View Controller architecture for all tools generated by XTools. Section 3.3 provides an example of a basic tool and section 3.4 provides an example of a meta-tool that shows how XTools can generate a tool that supports part of the XTools functionality. By implication,

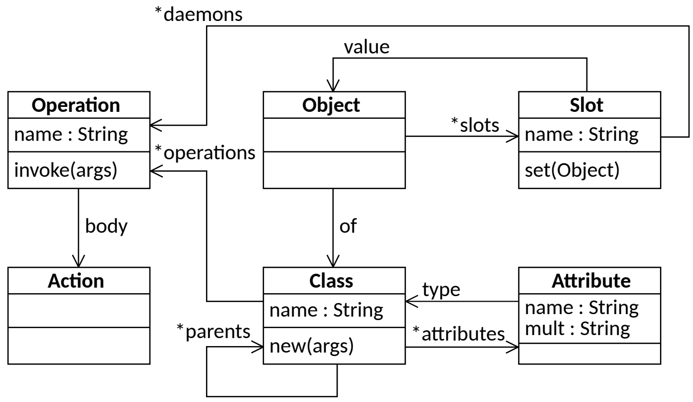

Figure 2: XCore

the basic tool can then be generated from the meta-tool.

3.1 XCore

Figure 2 shows the essential features of XCore that are used as the basis of the entire XModeler environment. Each of the classes shown in the XCore model are sub-classes of `Object` and instances of `Class`. Each of the edges on the diagram are instances of `Attribute` (associations are constructed in terms of combinations of attributes and constraints).

Several important aspects of figure 2 contribute to XModeler's ability to describe, reflect, modify and execute itself as follows. **State:** since all classes inherit from `Object`, all objects in the system reference slot objects that contain the object's storage. Operations for accessing and updating the fields of an instance of `Slot` are provided. Therefore, XModeler can reflect upon, and update, itself. **Behaviour:** the meta-model contains a class called `Operation` that described behaviour. The details of the expression and action language referenced by `Operation` has been elided, however each operation can be invoked on arguments. **Daemons:** operations can be registered with a slot so that each time the slot is updated the operations are called. Operations that monitor slots in this way are called *daemons* and can be used to implement a model-view-controller pattern and general purpose event handling. **Types:** objects have a link `of` to their class which contains a complete description of the structure and behaviour of the object. Note that since `Class` inherits from `Object` this property also applies to classes (meta-classes, *etc.*). Also, since classes are themselves objects, reflection can be applied to all parts of any XCore implemented system. **MetaTypes:** a class which inherits from `Class` is, by definition, a *meta-class* and that can redefined the operation `new` which uses the structure and behaviour defined in a class to create an instance. **MOP:** the operation `new` is part of the *meta-object protocol* (MOP) for XModeler that allows extensions of `Class` to affect all structural and behavioural aspects of a new language. The XCore MOP is implemented in terms of slot access and update, message passing, and object creation. Basic definitions are provided by XCore, however each of these may be redefined by sub-classes of `Class` making XCore arbitrarily extensible.

New tools can easily be added and integrated with all other tools in the system since there is a minimal reflexive interface defined in terms of a common representation for state and behaviour. New tools can be defined in XCore that manipulate any new form of object since it is always possible to reflect on the internal structure and behaviour of all objects. The MOP ensures that even if structure and behaviour is radically changed in a new XCore based language, a tool will always be able to send messages to objects, access the state of an object as though it was a named slot that can be updated via its name.

Daemons are the basis for an implementation of the MVC pattern which allows multiple tools to manipulate the same model elements: when one tool updates an object all observing tools are informed.

Together, these XCore features ensure that XTools is a meta-circular MDSE workbench. XTools uses XCore to define an executable tool model based on the features defined in section 2. Since XCore supports both structure and behaviour, the XTool meta-tool can build *both* the structure and behaviour of any tool. The MVC mechanism allows such tools to manipulate any instance of an XCore defined model. Since XTools is an XCore defined model, XTools can define a tool that builds XTools as described in the next section.

3.2 The XTools Language

XTools is an XCore defined language with support for diagrams, trees, forms, events and text. Figure 3 shows the classes of the XTools architecture for diagrams and forms (trees are implemented as sub-classes of `FormElement`)[1].

Two key tool types are defined: `DiagramToolType` and `FormToolType`, where other types of tools including trees and text editors are defined as types of `FormElement`. We show the structure of a diagram tool type in terms of node types and edge types as an example: form elements follow the same pattern.

When a tool is created, it is registered with its element and appropriate daemons are added to implement the MVC pattern. Event handlers may change the state of the object by adding new sub-objects. It is the responsibility of the handler to add daemons to the any newly added objects so that the MVC pattern is correctly maintained.

[1] Text-based language engineering is described in [Clark et al., 2015b, Clark et al., 2015a]. Syntax classes are provided for all the concepts of XTools such as diagram elements, tree elements and events.

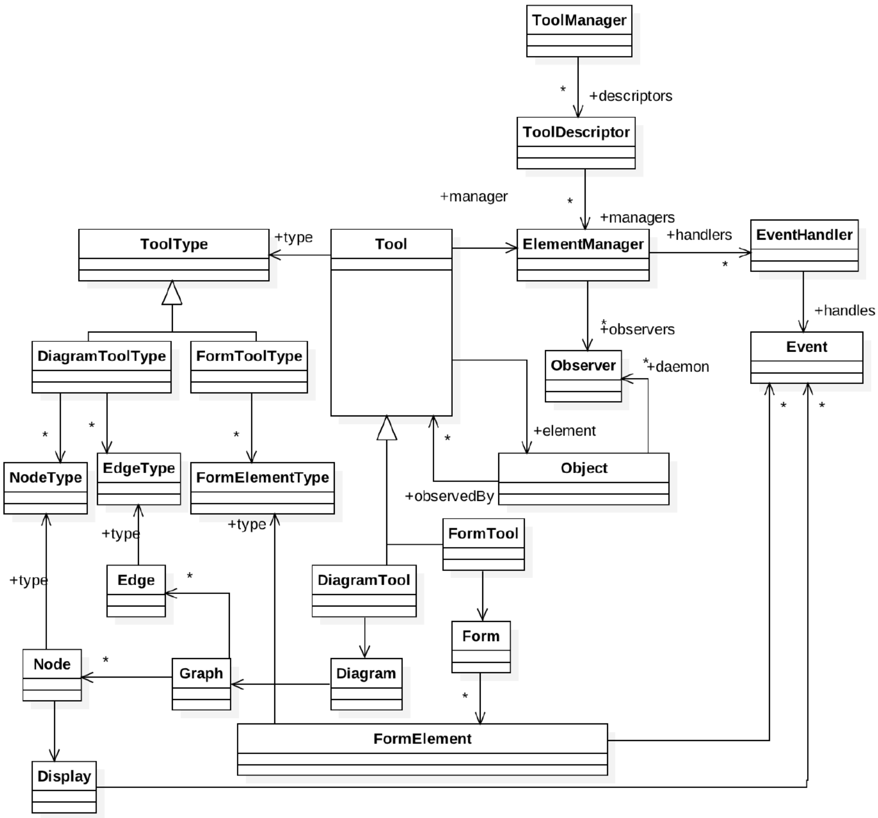

Figure 3: XTools Architecture

Since the languages of diagrams and forms is fixed, the type of events received by the event handlers is also fixed. For example events are generated on a diagram when a node is created, moved, resized, selected, *etc.* All of these events are processed by the tool's event handler. Table 1 shows the list of events generated by diagram elements. In each case the event name contains information that allows the handler to determine the type of the event.

3.3 An Integrated XTool

This section provides an example of a diagram tool in terms of its model, the diagram type, the event handlers and the model observers. The Critical Path Method (CPM) [White, 1969] is used to represent a plan in terms of its events and activities. The activities have a duration and are initiated and terminated by events that occur at particular times. We are interested in the scheduling of the activities and therefore associate earliest and latest times with the events.

Event Name Specification	Description
`New_`T`(t,o)`	A new node or edge of type T has been created. The event contains the new edge or node `o`.
T`_Removed(t,o)`	An existing node or edge `o` has been removed.
P`_Changed(t,d,s,s')`	Where P names a text element or an edge label that has been edited by the user. The event contains the display element `d`, the new text `s` and the old text `s'`.
E`_Target_Changed(t,e,n,n')`	Where E is the name of an edge type and the event contains the edge `e`, the new target node `n` and the old target node `n'`.
E`_Source_Changed(t,e,n,n')`	Where E is the name of an edge type and the event contains the edge `e`, the new source node `n` and the old source node `n'`.
`Add_To_`P`(t,d)`	Where P is a path to a starred container and the event contains the newly created display element `d`.
`Delete_From_`P`(t,d)`	Where P is a path to a starred container and `d` is the display element that has been removed.
P`_Clicked(t,o,i)`	Where P names a node, edge, label or display type and `o` is the element of that type that has been clicked, `i` is the number of clicks.
P`_Selected(t,o)`	Where P names a node, edge, label or display type and `o` is an element of that type that has been selected.
P`_Deselected(t,o)`	Where P names a node, edge, label or display type and `o` is an element of that type that has been deselected.
N`_Resized(t,n)`	Where N names a node type and `n` is a node of that type that has been resized.
N`_Moved(t,n)`	Where N names a node type and `n` is a node of that type that has been resized.
P`_ChangeTo(t,d)`	Where P names a disjunction of display element types and `d` is the new element that has been created.

Table 1: Diagram Event Specifications

An algorithm that consists of a forward and backward pass is used to propagate earliest and latest times for events based on the dependencies between actions and their duration. The difference between the two event times represents the *slack* in the project plan: a project with 0 slack is high-risk since a delay in any activity will delay the completion date of the project.

The CPM can be supported by a tool that constructs a graph of events and activities represented as nodes and edges respectively. Duration and times can be added as labels on the graph and the earliest and latest times can be constructed by functionality that is exposed via a menu. The model to be manipulated by a diagram tool is shown in figure 4. The tool creates a graph whose nodes are labelled with events and whose edges are labelled with activities. A graph supports operations: `roots()` that returns a set of nodes that are not the target of an edge; `terminals()` that returns a set of nodes that are not the source of any edge; `predecessors(n)` that returns the set of all nodes that are the source of an edge that has `n` as a target; `successors(n)` that returns the set of all nodes that is the target of an edge that has `n` as the source; `edgesBetween(n1,n2)` that returns the set of all edges with source `n1` and target `n2`.

The CPM associates an event with the latest and earliest times that the event can occur. These times are calculated based on the activities that terminate with

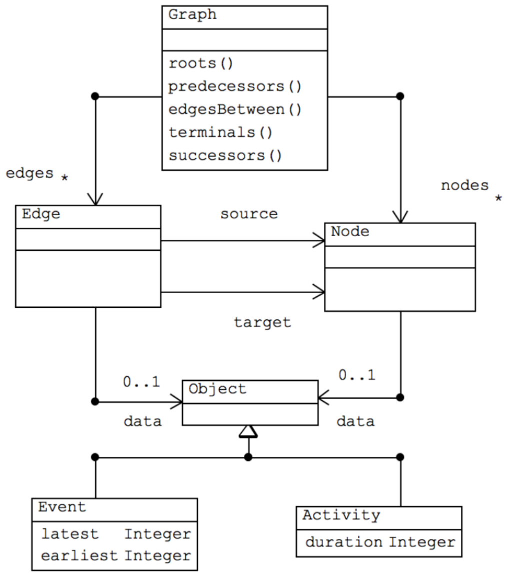

Figure 4: Meta-model of concepts in the CPM Diagram Language

the occurrence of the event. Often the earliest and latest times are the same, however where there are concurrent activities, events may occur within a time window: the *slack*.

Figure 5 shows a diagram XTool that has been generated based on the model shown in figure 4. The XTool is shown in the upper right panel and integrates fully with standard XModeler tools: an editor on the left and a property editor below. The diagram editor allows a CPM model to be constructed and supports some basic functionality to calculate the critical path.

3.4 Meta-Circularity

Meta-circularity (including features such as *reflection* and *adaptation*) provides a means for a system to reason about its own data and behaviour and to influence the semantics of these features [Bershadsky et al., 2018]. The previous sections have described the basic XTools architecture and provided a simple example of a diagram XTool. The meta-features described in Sect. 3.1 ensure that: (1) all XTools fully integrate with all other tools provided by the XModeler ecosystem, and (2) XTools is capable of defining an XTool-metatool.

The key to achieving a meta-circular XTool is creating an instance of figure 3 where the appropriate types (node-type, edge-type, form-type *etc.*) are in one

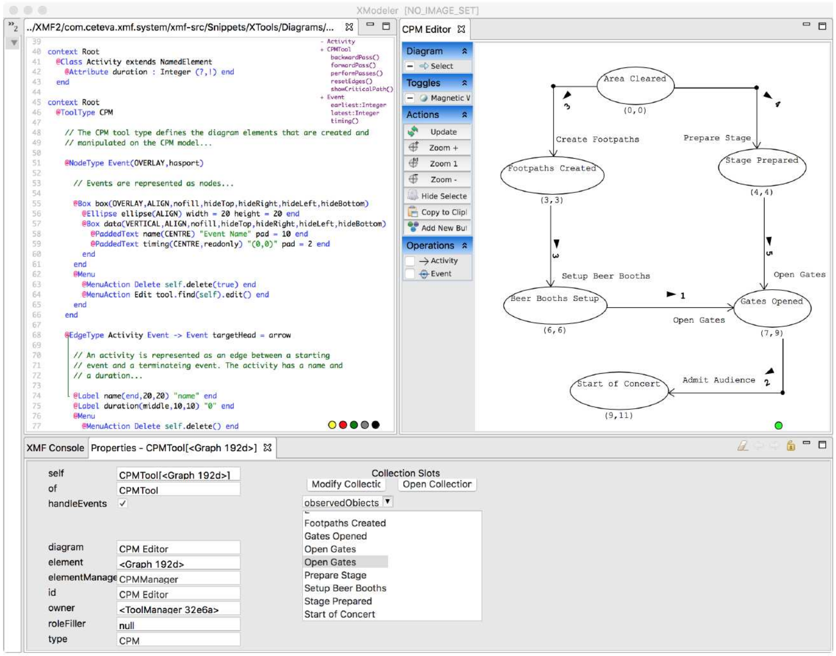

Figure 5: Example Critical-Path Model in an XTools Diagram Editor

to one correspondence with a meta-model defined in XCore. The resulting tool will then create instances of meta-classes.

Note that it is possible to create an instance of figure 3 where the type classes are in correspondence with themselves, thereby creating an XTool that creates XTools. The event handling classes and their associated behaviours are just XCore defined classes so an XTool can create XTool behaviour. Therefore, XTools is fully meta-circular because it can create an XTool that supports all of the XTools functionality. Note also, that an XTool can be created that supports a sub-set of the XTools functionality (for example a diagram editor for diagram editors). In addition, an XTool can be created that supports a mixed mode of standard and meta-classes.

Figure 6 shows an XTool-metatool defined as an XTool tree editor on the bottom right of the screen shot. This XTool allows the user to create a diagram XTool starting with a root tree element that corresponds to a new XTool type and attach it to an arbitrary class. After creating the root node, the user is guided through a number of steps that create tool bars, node types and edge types together with their event handlers corresponding to the event types defined in Table 1. The metatool uses introspection on the class to guide the creation of the diagram editing features and the event handlers.

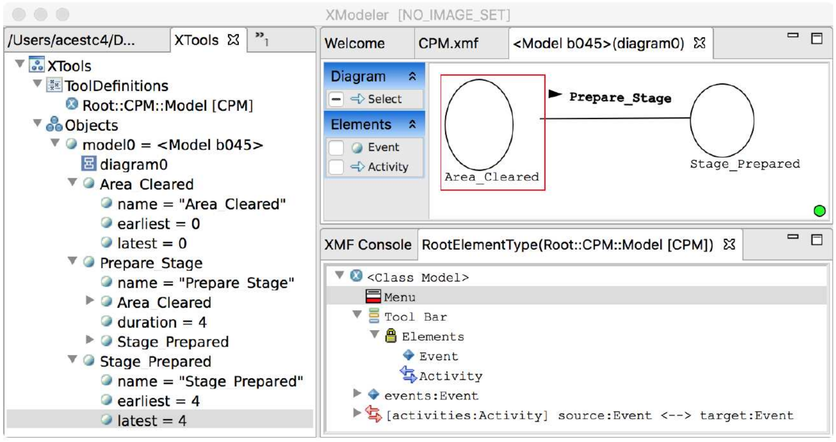

Figure 6: An XTool for XTool Creation

Once the XTool-metatool is created it can be instantiated any number of times within XModeler with respect to a particular instance of the controlled class. Figure 6 shows an instance of the meta tool on the right where the event nodes and activity edges have been created using the palette buttons.

4 Established Approaches

The previous section has provided a conceptual overview of XTools and outlined how it can generate a domain specific language (CPM) and a meta-language for creating XTools. As such XTools is a language workbench with reflexive capabilities satisfying the requirements defined in section 2. This section reviews leading MDSE workbench technologies prior to comparing them against the requirements in section 5.

4.1 Eclipse Graphical Modelling Framework (GMF)

The Eclipse Graphical Modelling Framework (GMF) [Gronback, 2009] is a meta-modelling environment for creating graph-based diagram editors. It is based on the Eclipse Modelling Framework (EMF) [Steinberg et al., 2009, Eclipse Foundation, a], which provides meta-modelling capabilities and basic model editor generation features to create a tree-view editor for EMF model instances. Using EMF's meta-modelling language Ecore as a common basis, GMF adds additional models that allow for specifying concrete diagram syntaxes for concepts in Ecore meta-models, as well as a code generation infrastructure which takes concrete diagram syntax specifications and generates diagram editors from it. The generated editors are subsequently available as Eclipse plugins, and the generated source code of the plugins can additionally be edited to implement specific features not provided by the default editor.

Five separate model types are involved in the generation of a diagram editor and are used to streamline the development of GMF's internal diagram editor code generation templates rather than to provide domain-specific abstractions for the purpose of visual language design.

Despite its conceptual limitations, GMF has become one of the most widespread diagram editor creation tooling environments and is freely available as a standard set of open-source Eclipse plugins that integrate with the EMF framework and auxiliary technologies such as model transformation engines and code generator components.

Internally, GMF is implemented as a front-end for the Eclipse Graphical Editing Framework (GEF) library, which is a non-model-based, traditional Java API to supplement the implementation of diagram editors for Eclipse. GMF translates central API concepts of GEF to model concepts in its five interrelated modelling languages, and provides code generation functionality to assemble their configurations. Close connection to the underlying GEF API ties GMF to predominantly implementation-level abstractions.

Conceptually, the approach chosen in GMF to describe concrete syntax for modelling languages suffers from another major shortcoming: Different types of visual elements, such as line connectors and nesting containers, can only be used in combination with fixed meta-model concepts.

The generative approach of GMF, which uses code generation during development time to provide an editor plugin to be run in a separate Eclipse environment, supports flexible modification of the generated editor functionality. Systematic extensions to the GMF generation process exist that re-apply source code modifications after changes to the editor specification and subsequent regeneration of the editor code [Gulden, 2009].

4.2 Graphiti

Graphiti [Brand et al., 2011] was originally developed as an alternative to GMF, and later contributed as an open-source component to the set of openly available Eclipse extensions. It also is based on EMF, but unlike GMF, the specification of a visual language is not provided by specialised models, but purely by programming against an API that implements a default diagram editor that makes use of a standard "boxes-and-lines"-notation style for entities and relationships in a model. The types of entities that are available, their attributes, and relationship types are configured programmatically by defining a diagram type agent class as a subclass of an abstract super-class provided by the Graphiti API.

Given the programmatic nature of Graphiti, no code generation is involved when creating a diagram editor. However, the configuration mechanism remains on a very low level of abstraction and requires each bit of editor functionality to be manually registered as a separate code fragment for each model element.

Although Graphiti claims to be "an alternative to GMF" [Brand et al., 2011], the differences to GMF in use are significant. As the framework offers functionality on a low level of syntactic diagram rendering and editing only, the burden is on the language developer to find possible abstractions that reduce redundancies and allow an efficient specification of visual languages.

4.3 Eugenia

Eugenia [Kolovos et al., 2017, Rose et al., 2012] is a wrapper around EMF and GMF that integrates the Ecore, Genmodel, Graphmodel, Toolmodel and GmfModel formats by offering a textual specification language that supports concepts from the original GMF framework in one single language. A generator is attached to it, which extracts information from the textual specification into the separate model formats for further processing with the original EMF / GMF infrastructure.

For developers who are already familiar with GMF, Eugenia provides an efficient shortcut for creating the required models for that framework. However, Eugenia misses the chance to improve the semantic expressivity of GMF, since it does not provide means for simplifying the GMF specifications in conceptual terms, *i.e.*, no higher-level abstractions are offered.

4.4 Sirius

Sirius [Viyović et al., 2014] is the most recent member among EMF extensions for specifying concrete model syntax. Like GMF and Graphiti, it integrates seamlessly with the Eclipse tooling environment and the Eclipse Modeling Framework. Sirius also uses a model-based configuration approach, in which a developer edits a tree-structured configuration model from which the diagram definition is derived. There are two major differences between GMF and Sirius. Firstly, Sirius does not use code-generation to transfer the configuration model to executable code. The model rather is interpreted by a run-time component which makes it easier to modify or extend the diagram description in an interactive process without the need to re-generate the diagram editor code after each round of changes.

A second and more significant difference to most other concrete syntax specification approaches is that Sirius supports visual examples in real-time during language development, which gives an immediate visual feedback to the developer about how the specified visual syntax looks like.

Sirius also includes non-diagrammatic model representations: table views (matrices) and tree views (comparable to EMF's tree view representation) and therefore can be seen as a tooling environment for domain specific *workbenches*.

Overall, Sirius appears to be the most advanced Eclipse-based visual language design approach. Its interpretative approach allows for fast development cycles, and the simultaneous design of the language specification and a prototypical model instance appears a fruitful approach for visual language design, as it gives immediate visual feedback to the developer of how the specified visual language looks like.

Sirius is available as an open-source solution via Eclipse update sites, and bundled as a commercial integrated modelling tool environment under the name Obeo Designer [Obeo, 2020].

4.5 Eclipse EEF

With the Eclipse Editing Framework (EEF) no diagram representations are defined, but forms are specified which make model content visible and editable. When EEF is used in combination with other diagram definition approaches, this also supports the idea that multiple diagram and non-diagram views of a model together make up a complete application.

EEF supports a mapping from attributes and associations to form patterns. No higher conceptual abstractions about the purpose or process of using forms are leveraged by the approach. As a consequence, any advanced functionality with respect to the dynamic interaction with forms requires program code fragments, *e.g.*, for form field validation.

A more advanced conceptualisation of forms could support *form states* that change while using the form, and that potentially influence its visual appearance. In addition to state-based approaches [Harel, 1987], the use of declarative process modelling languages [Marquard et al., 2016] appears to point into the direction of a more elaborate way to conceptualize forms, which, however, goes beyond the capabilities offered by EEF.

4.6 MetaEdit+

The meta-modelling environment MetaEdit+ [Tolvanen and Rossi, 2003, Tolvanen and Kelly, 2009] offers an integrated approach for domain-specific language creation and application. The tool suite comes with a proprietary infrastructure and is not based on the Eclipse Modeling Framework (EMF). Instead, all functionality is realised by two interrelated desktop applications: MetaEdit Workbench for defining modelling languages, and MetaEdit Modeler for editing model instances and running transformations. The first version of the tooling environment dates back to 1995, which by today has made MetaEdit+ a matured approach offered as a commercial product.

Languages in MetaEdit+ are defined by specifying the desired entity types, attributes, and relationships through a form-based user interface. No explicit meta-model is created by default, instead, all characteristics of diagram graphs, entity-types, properties, connectable ports, and relationships are entered non-graphically in a traditional user interface. Consequently, this language specification approach is called GOPPRR, which stands for Graph-Object-Property-Port-Role-Relationship. Both language definitions as well as model instances are stored in a central repository, not as files. This makes MetaEdit+ suitable for large and distributed development projects.

The environment follows an interpretative approach, *i.e.*, no code generation is involved when creating model editors. Instead, the MetaEdit Modeler tool interprets available language definitions at run-time to offer according editing functionality. Changes that are made to the language definition in the MetaEdit Workbench are directly reflected in the MetaEdit Modeler. Besides the built-in behaviour of the modelling tools, there is little possibility for language designers to customize the resulting editor.

4.7 Melanee

The research prototype Melanee [Atkinson and Gerbig, 2016] primarily aims at providing a modelling environment for multi-level conceptual modelling [Frank, 2014, Odell, 1994, Atkinson and Kühne, 2001]. The tool works on the basis of an object-oriented modelling language that supports classes, objects, attributes, and relationships. These basic language elements are enhanced with additional features for multi-level conceptual modelling and visual representation of models. When creating multi-level models with Melanee, class diagrams for each abstraction level are shown vertically stacked, one above the other, on a diagram canvas. Each diagram represents one abstraction level, with the highest level shown on the top. By default, the model elements in each lane are displayed in a UML-like notation, *i.e.*, classes and objects are represented as rectangular boxes, with a name label in the top-middle of the box, and a compartment for attributes. Operations are not supported. Relationships appear as line connectors by default. For displaying more details about a relationship, the notation of a relationship can be switched to an entity-style mode, in which the concept of the relationship appears as an entity symbol on the diagram canvas, linked through line connectors to the connected entity types.

In addition to the UML-like notation of model elements, model elements can be switched to a DSL syntax that can be specified for each model element by assigning graphical primitives or images to the defining entity in a conceptual model on an abstraction level above the currently edited one. Models that originate from a multi-level hierarchy of classes can also make use of an aspect-oriented modification of the visual representation they inherit from a higher level of class abstractions.

Defining a visual notation like this can be considered a stereotyping of the original UML notation, as the existing notation becomes adapted to a domain-specific case. This is easy to specify, but provides little flexibility when other visual metaphors apart from nodes and connectors are to be used in a visual language.

The tool has specifically been designed for working with multi-level conceptual model hierarchies. As a consequence Melanee does not support creating workbenches with multiple synchronised model views.

4.8 Xtext

A prominent representative of a textual modelling approach is Xtext [Grönniger et al., 2014, Eclipse Foundation, c]. Based on Enhanced Backus-Naur-Form (EBNF) grammar specifications [Nijholt, 1988], the framework supports textual languages and can automatically generate parsers which transform textual model instances into parsed object trees in the format of EMF [Steinberg et al., 2009] instances. The resulting integration with the standard tooling environment of Eclipse provides a powerful alternative to meta-modelling for efficiently creating (small and medium sized) domain specific textual modelling languages. Xtext exclusively focuses on textual model views and does not support other types of representations.

4.9 Meta Programming System MPS

The MPS meta programming system [Campagne, 2014, Voelter, 2013] provides a *projective* approach to language engineering. This means, a modelling language is not defined via a meta-model or a grammar to create a single model editor that uses one consistent editing paradigm for the entire modelling language. Instead, languages are composed of multiple partial editor definitions, which each may associate different so-called “cells” in a graphical user interface (GUI) with content of a model that is being edited. The GUI elements and the model contents are synchronised using a model-view-controller (MVC) [Reenskaug, 2007, Mahemoff and Johnston, 1999] pattern. This makes sure the GUI elements are updated according to the contents of the object tree whenever changes to the objects occur, so the user of a model editor always sees the current contents of the objects, and whenever the user edits contents using the GUI elements, changes are propagated back into the object tree of the model.

Such a projective approach supports complex structured model editors, because the GUI elements that provide the editor interface can be chosen from multiple kinds of interaction elements, comparable to the design of a traditional application GUI. This includes structured text, tables, sliders, diagrams, and other elements. The language definition associates element types in a tree model with suitable GUI representations, and the projective editor environment makes sure that the MVC behaviour is applied to the individual elements. As a downside, working with such editors requires modellers to be more aware of the internal object tree structure than with single-paradigm model editors. For example, when textually editing a model in a projective editor, the language elements are individually addressed as sections of the text during editing, instead of letting the user freely type text that is subsequently transformed into an object tree structure using a parser.

5 Comparison and Evaluation of the Approaches

The general vision of a model-based approach to the definition of Software Engineering tools is discussed in [Clark et al., 2013]. A fundamental understanding of basic properties of modeling tool support with inherently multiple views means is established in [Goldschmidt et al., 2012]. While not all concepts discussed in that work are one-to-one reflected in XTools, it shares the notions of key terms such as *view*, *view type*, and *viewpoint*.

In [Pfeiffer and Pichler, 2008], a text-based comparison framework is developed that is based on dimensions that are grouped into criteria concerning the modelling *language definition*, the model *transformation* capabilities, and *tooling* aspects of language workbench environments. The framework is subsequently applied to the language environments openArchitectureWare (oAW) [Efftinge et al., 2008], the Meta Programming System (MPS) [Campagne, 2014, Voelter, 2013], MontiCore [Krahn et al., 2010], and four other approaches, each in a version that was current when the examination was published in 2008.

The case study in [El Kouhen et al., 2012] evaluates different language engineering workbenches along the example case of implementing a simplified version of the Business Process Modelling and Notation (BPMN) language [Weske, 2012, Dumas et al., 2013] and its visual diagram syntax.

With emphasis on conceptual language definition capabilities realised by different meta-modelling approaches, [Kern et al., 2011] examines the expressivity offered by meta-modelling approaches from 6 different tooling environments, among them ARIS [Scheer and Schneider, 2005], the Eclipse Modelling Framework (EMF), and MetaEdit+. The comparison provides a comprehensive overview of the differences between language specification approaches, but operates with a narrow perspective by only focusing on conceptual language specification selected from a few approaches.

Quantitative measures of language workbenches are compared in [Kelly, 2013] including *feature coverage, lines of code, user satisfaction, time, and cost*.

The annual Language Workbench Challenge is described in [Erdweg et al., 2013] and [Erdweg et al., 2015]. It involves the construction of domain-specific languages with different language workbenches provided by the participants. However, many prominent contemporary approaches are not included.

In summary, existing pieces of work on comparing meta-modelling environments and language workbenches take different starting points compared to that described in this article, such as means for conceptual language definition [Kern et al., 2011], explicit restrictions toward non-visual usage paradigms [Pfeiffer and Pichler, 2008], or an emphasis on tooling aspects [El Kouhen et al., 2012]. The support for the definition of visual representations of models and subsequent user interaction seems not to have been in the focus of existing comparative analyses.

When developing our framework we concentrate on capabilities offered by different language engineering workbenches that contribute to the reflexive tooling.

5.1 Comparison Criteria

Views provide access to the model and include *Diagram view* (see *RQ-1*) consisting of nodes and line-connectors. *Form views* display model instance content using elements such as text-fields, lists, check-boxes, and radio-buttons. A special kind of form view is a *tree view*, which extends the notion of a list by the additional concept of a hierarchy. Language engineering workbenches may offer form views individually or in combination with diagram views, since diagrams typically do not offer capabilities for editing a complete set of object properties. A *Matrix view* can be used to visualize and edit relationships. A matrix view consists of a square area with two axes, along which object instances are listed. The objects are represented as textual labels, potentially in combination with graphical icons to indicate an object's type and / or status. These criteria are reflected second in the comparison (Table 2) and corresponds to requirements *RQ-2* and *RQ-4*.

A *Textual view* may also be offered by language engineering workbenches to represent model instances. Whether such a view type is provided by a language engineering workbench, is indicated with a marker at the respective position in the comparison table (Table 2). The availability of a textual view directly fulfils requirement *RQ-3*.

Approaches differ in terms of view integration and synchronization. The criterion *Integration of views* expresses this ability (see *RQ-8*). We consider a tool to support view integration if it is able to display several views in parallel showing different synchronized perspectives on the model content.

Approaches which make use of *editor domain abstractions* are indicated in the row labelled "Editor domain abstractions above implementation level" in Table 2. The comparison criteria related to editor domain abstractions contributes to requirement *RQ-5* and requirement *RQ-10*.

We also consider whether language engineering workbenches provide *model based notation specification* or *text based notation specification* options for defining representation and interaction features. In the evaluation framework, a marker is set for the first criterion, if any non-textual specification mechanism is made available by a language engineering workbench, and accordingly for the second criterion, if a text-only specification is possible.

If *run-time notation editing* is available a marker is set for the according language engineering workbench approach. If no run-time editing is made available, the development of model instance editors implies the editor to be re-initialised each time a change to the configuration is made. Availability of run-time notation editing capabilities contributes to satisfying requirement *RQ-6*.

We consider three criteria regarding the way a visual concrete syntax is specified. Many approaches support *symbol definition via code or model*. Language engineering workbenches which offer such a specification mechanism are indicated with a marker in the row that corresponds to this criterion. In contrast, some language engineering workbenches import graphics from external sources where a marker is set in the row *External symbol definition*. Some approaches also provide tool support for *visual editing of symbols* inside the language engineering workbench. A tool that offers such functionality is assigned a marker in the corresponding row and is satisfying requirement *RQ-6*.

A fundamental distinction in the way model instance editor specifications are processed lies in the way an executable model editor artifact is created. Two mutually exclusive modes arise: *code generation* or *run-time interpretation* of model instances.

Dynamic adaptation to model content, *e.g.* adapting a symbol's context menu according to underlying model elements' states is a relatively advanced feature that has a marker set in the *dynamic symbols depending on model state* row satisfying the requirement *RQ-6*.

The use of *form elements in diagrams* in the concrete model syntax can be considered as one of the most advanced concrete syntax features of model instance editors. Language engineering workbenches which are able to provide these capabilities are assigned a marker in the corresponding row and satisfy requirement *RQ-6*.

The semantics requirement *RQ-7* is an inherent part of the underlying conceptual model and any model instance editor that does not respect semantic rules is considered faulty. Therefore, these aspects intentionally are not part of the comparison in this article, and the reader is referred to other sources [Kern et al., 2011, Erdweg et al., 2015, Vujović et al., 2014]. In addition, requirement *RQ-11 – Extension* is not reflected by any of our comparison criteria.

	GMF	Graphiti	Sirius	Eugenia	EEF	MetaEdit+	Melanee	XText	MPS	XTools
Diagram view	✓	✓	✓	✓	–	✓	✓	–	✓	☆
Form view	★[2]	✓[2]	★[3]	✓[2]	★	✓[2]	✓[2]	–	★	★
Matrix view	–	–	✓	–	–	✓	–	–	✓	✓[4]
Textual view	–	–	–	–	–	–	✓	★	✓	★
Integration of diagram / form / matrix / textual views	–	–	✓	–	–	–	✓	–	✓	☆
Editor domain abstractions above implementation level	–	–	✓	–	–	–	–	–	–	★[4]
Model based notation specification	✓	–	★	–	✓	–	✓	✓[5]	★	★
Text based notation specification	–	✓[6]	–	✓[7]	–	–	–	★	–	★
Run-time notation editing	–	–	✓	–	–	✓	✓	–	–	✓
Symbol definition via code or model	✓	✓	✓	✓	–	–	✓	–	✓	✓
External symbol definition	✓	–	✓	–	–	✓	✓	–	✓	✓
Visual editing of symbols	–	–	–	–	–	✓	–	–	–	–
Code generation	✓	–	–	✓	–	–	–	✓	✓	–
Run-time interpretation	–	–	✓	–	✓[8]	✓	✓[9]	–	–	✓
Dynamic symbols depending on model state	✓[10]	–	–	–	–	✓	✓	–	✓[11]	✓[12]
Form elements in diagrams	–	–	–	–	–	–	–	–	✓	✓

Symbol legend:
✓ available
★ available and self-reflexively used for language specification
☆ available and self-reflexively usable for language specification
– not available

Table 2: Comparison of the Examined Approaches

5.2 Comparison of Language Engineering Workbenches

Table 2 contains an evaluation of each of the language engineering workbenches that have been introduced in Sect. 4. Each criterion is marked with a "✓" if it is covered by the approach in question, or with a "–" if not. A "★" symbol indicates that the language engineering workbench offers a feature and at the same time uses it as part of its language specification mechanism, *i.e.*, the criterion is reflexively applicable and can be used to model the language engineering workbench itself. A "☆" indicates that a workbench has the capability of using a feature reflexively, but does not use it in its own definition.

Most candidates offer only a limited set of model representation and interaction views. Approaches that fall into this category are GMF, Graphiti, and Eugenia. EEF and Xtext also provide one view only, which are a form view and a text view, respectively.

[2] Default properties view.
[3] By integration of EEF.
[4] By domain-specific language extensions using *syntax classes*.
[5] By import of Ecore meta-model.
[6] By Java program code.
[7] Uses Emfatic (`.emf`) text representation of the GMF model family.
[8] Code generation until version 1.6, since version 1.7 run-time interpretation.
[9] Run-time interpretation by external tools only.
[10] By modification of generated code.
[11] By manual implementation of `javax.swing.Icon`.
[12] By attaching listeners ("daemons") to model elements and changing visual appearance accordingly.

The largest number of criteria are addressed by Sirius, MetaEdit+, MPS, and XTools. These representatives can be distinguished according to their primary purposes. MetaEdit+ and MPS originate from model-driven software engineering (MDSE) approaches. Although following entirely different realisation paradigms, both approaches have initially been created to ease programming and to allow gaining a higher level of abstraction in the specification of software systems. MetaEdit+ uses, among others, visual diagram elements to provide such abstractions. MPS has extended the notion of a graphical user interface to a projective modelling approach [Voelter and Lisson, 2014]. Both of these approaches focus on abstracting over software structures for code generation [Kelly and Tolvanen, 2008, Voelter, 2013], rather than the meaning of concepts.

XTools provides capabilities for a systematic integration of multiple modelling perspectives on the language definition level. Sirius and MPS offer this, too, by a variety of model representation and interaction views both during language specification and when using languages. As a consequence, Sirius, MPS and XTools primarily follow the idea of using composed model editors like application workbenches, and while MPS is oriented toward model-based software development, Sirius and XTools offer facilities to model language workbenches. Sirius makes use of the standard meta-modelling capabilities offered by Ecore, which is the central meta-modelling language of the Eclipse Modelling Framework (EMF) for expressing the abstract syntax and semantics of conceptual models although operations are limited to their signatures. XModeler meta-models do not require code generation (the meta-model elements are represented internally like any other programming language constructs). Models can be interpreted by the XOCL language framework or can be compiled *in-situ* to the XModeler VM (both the interpreter and compiler are written in XOCL). Therefore, the specification of business logic with behavioural language constructs is integrated with XCore meta-models.

The advanced meta-model specification capabilities and full integration with XModeler makes XTools compare favourably with Sirius in terms of realising the overall vision of creating domain-specific modelling tools in a model-driven way.

XTools offers three different types of built-in views on models. The set of model view types offered by Sirius covers four basic types of views: graph diagrams, tree-views, forms, and a matrix view. Matrix views display relationships among model content in a 2-dimensional structure. For specifying form-based views on models, Sirius integrates the features of the EEF framework, which also is available as a separate component for the Eclipse Modelling Framework to specify form-based property views for conceptual models (Sect. 4.5). The set of basic view types has been enhanced by additional components such as a sequence diagram view, which corresponds to a UML sequence diagram type for showing procedural model content. The specification of model views is performed in Sirius by editing a tree model structure. This provides an abstraction from writing code, at the price of lesser flexible editing operations.

Both XTools and Sirius support model editor specification at run-time, without an intermediate step of code generation and / or compilation.

Sirius uses standard EMF notification mechanisms that manually implement view synchronization based on typical event listener mechanisms similar to those used throughout Java APIs. Architectural support for integrating views, however, rarely is available in existing language engineering workbenches perhaps because it requires subtle adjustments of transaction handling and consistency management [Brun and Köhnlein, 2017]. XModeler makes use of the listener mechanisms of the underlying XOCL language called *daemons* (Sect. 3).

By introducing syntax classes for describing view interrelationships with a domain-specific language, XModeler gets closer than Sirius to the idea of a language engineering workbench which serves to specify entire (business) applications such as MDSE tooling environments as a combination of conceptual models and a set of interrelated model views.

The range of generalised functionality that XTools uses through the XModeler infrastructure comes with the drawback of many detailed configuration tasks that in general have to be performed on the program code level. The XTools meta-tool (Sect. 3.4) addresses this by making the construction of modelling views easier.

5.3 Comparison of Reflexive Capabilities

MDSE language workbenches offer features that allow the user to create and deploy languages and tools that target specific domains. Since MDSE tooling is itself a domain, it makes sense to analyze the workbenches with respect to *reflexive* features, *i.e.*, the ability to define language features that would enable the workbenches to build variations of themselves and thus make use of a meta-circular architecture (Sect. 3.4). In many cases, established workbenches were not designed to build themselves or the capability to reflect and build meta-variations is undocumented.

When a language engineering workbench makes use of form views for entering the configuration of a language workbench, and these form views are defined by means of the language engineering approach itself, a "★" marker has been set in the comparison matrix in Table 2.

GMF makes use of tree-based EMF model editors to assemble its configuration for visual modeling languages. As GMF is an extension to EMF and implicitly includes all its features, this way of specifying language definitions constitutes a self-reflexive use of GMF's modeling workbench specification mechanisms.

Sirius makes use of self-reflexively defined tree editors in an explicit way, so that its workbench definition language explicitly contains constructs which allow to define their own configuration tool in a reflexive way.

EEF also uses EMF-based tree editors for defining its property form configurations that offer known EEF features and the EEF look-and-feel, thereby providing a way for EEF to define itself.

The interface of the MPS modeling workbench for defining languages makes use of a projection-oriented approach that supports a combination of a text-based specification and configuration options via user interface widgets that can be used to configure MPS itself.

The tree editor widget of the model-based language specification approach offered by XTools realizes a reflexive use of the language specification mechanism since the model-based XTools specification environment has been specified using XTools itself (see Sect. 3).

XText is based on a reflexive use of its textual language specification mechanism, since the grammar specification for XText-based languages is internally handled as a model, and it is entered with an Xtext-generated textual model editor. A comparable approach is taken by XTools, which with the help of *syntax classes* defines individual grammar fragments for parts of a textual model instance. The format for textual XTools configurations makes explicit use of syntax class definitions providing a reflexive specification mechanism.

To our knowledge, the only language engineering workbench that provides a model-based approach to engineering workbenches is XTools. All concepts for describing language engineering workbenches can be expressed and be combined with the existing set of conceptual abstractions for model editor design. However, while XTools provides high level reflexive meta-concepts it does not come with a library of workbench elements. The use of XModeler and XTools to define reusable reflexive workbench features is an area for further work.

A "★" in the "Model based notation specification" row in Table 2 coincides with one in the "Form views" row, because in all examined approaches the self-reflexive use of model based notation specification happens with tree-based model editors that have associated property forms.

Some approaches provide a textual specification mechanism for language engineering workbenches. A "★" in the row "Text based notation specification" in Table 2 coincides with one in the row "Textual views", because a self-reflexive use of a textual specification mechanism implies that textual views are made available by the approach.

6 Conclusion and Future Perspectives

This article has contributed to the field of meta-circular MDSE workbenches by reviewing established technologies in this area, identifying a collection of requirements that should be provided in order to be both MDSE *and* reflexive, and comparing the technologies against the requirements.

Unlike other tools, The tool XModeler was designed to be meta-circular and has achieved this by bootstrapping modelling languages and tools from a small core. XModeler validates the claim that it is an MDSE meta-circular workbench by providing a sub-framework called XTools and then using parts of XTools to define a model-based XTools meta-tool. The comparison of MDSE frameworks with respect to meta-circular criteria is shown in Table 2. This table can be viewed in two ways: features required for modelling and features required to model an MDSE workbench. In the first case, the table shows that established workbenches offer a variety of features for modelling, but, as shown in the second case, often these features cannot be used in a reflexive way.

Given the design motivation of XModeler, it is perhaps not surprising that XTools offers the greatest number of modelling features that can be applied to itself. However, XTools is not complete with respect to meta-circular features and is a research prototype with proprietary technological approaches. As a consequence, it cannot be considered an optimal platform for industry-standard model-driven software engineering development projects. Other tools can individually be identified as best-of-breed for tasks such as conceptual language definition or graphical visualization of models. In contrast, XModeler serves as an example for an architectural style which hopefully will influence the future development of language engineering workbenches in general.

Our claim in this article is that MDSE workbenches should aim for meta-circularity since this achieves several desirable benefits. However, there are a number of disadvantages to meta-circularity that can be viewed as outstanding research challenges in this area: static type-checking is difficult; execution of a meta-circular core may jeopardise performance; the maintenance of tools written using a meta-modelling approach can be a problem; there are no standards for meta-circular tool interoperability.

Although XModeler and XTools is not complete with respect to the requirements for MDSE meta-tooling described in this article, we claim that it provides a contribution to the field in terms of its design aspirations. We propose that next generation MDSE should be designed to be reflexive and meta-circular in the sense that the modelling features of the tool can at least be used to build itself and variations thereof.

Although XTools provides unrestricted access to executable meta-levels through the facilities of XModeler, there is significant scope for adding structure and semantics to the definition of DSL tools in this way. For example, several meta-levels can co-exist in order to link meta-classes, classes, and their instances. Such multi-level modelling [Frank, 2014, Odell, 1994, Atkinson and Kühne, 2001] is necessary to represent tool-types, tools, and their instances. However, from an engineering perspective, it becomes difficult to manage the levels without tool support for enforcing type distinctions, and this is an area for further language development.

References

[Atkinson and Gerbig, 2016] Atkinson, C. and Gerbig, R. (2016). Flexible deep modeling with Melanee. In Reimer, S. B. U., editor, *Modellierung 2016, 2.-4. März 2016, Karlsruhe - Workshopband*, volume 255, pages 117–122, Bonn. Gesellschaft für Informatik.

[Atkinson and Kühne, 2001] Atkinson, C. and Kühne, T. (2001). The essence of multilevel metamodeling. In Gogolla, M. and Kobryn, C., editors, *UML '01 Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 19–33, London. Springer UK.

[Bershadsky et al., 2018] Bershadsky, A. M., Bozhday, A. S., Evseeva, Y. I., and Gudkov, A. A. (2018). The mathematical model of reflection for self-adaptive software. In *2018 9th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–5. IEEE.

[Beydeda et al., 2005] Beydeda, S., Book, M., and Gruhn, V., editors (2005). *Model-Driven Software Development.* Springer, Berlin Heidelberg.

[Brand et al., 2011] Brand, C., Gorning, M., Kaiser, T., Pasch, J., and Wenz, M. (2011). Development of high-quality graphical model editors. *Eclipse Magazine*, (1).

[Brun and Köhnlein, 2017] Brun, C. and Köhnlein, J. (2017). Integrating Xtext and Sirius: Strategies and pitfalls.

[Campagne, 2014] Campagne, F. (2014). *The MPS Language Workbench: Volume I.* The MPS Language Workbench. CreateSpace Independent Publishing Platform.

[Clark et al., 2013] Clark, T., France, R. B., Gogolla, M., and Selic, B. V. (2013). Meta-Modeling Model-Based Engineering Tools (Dagstuhl Seminar 13182). *Dagstuhl Reports*, 3(4):188–226.

[Clark et al., 2015a] Clark, T., Sammut, P., and Willans, J. S. (2015a). Applied metamodelling: A foundation for language driven development (third edition). *CoRR*, abs/1505.00149.

[Clark et al., 2015b] Clark, T., Sammut, P., and Willans, J. S. (2015b). Superlanguages: Developing languages and applications with XMF (second edition). *CoRR*, abs/1506.03363.

[Clark and Willans, 2013] Clark, T. and Willans, J. (2013). Software language engineering with XMF and XModeler. In Mernik, M., editor, *Formal and practical aspects of domain-specific languages: recent developments*, pages 311–340. IGI Global.

[da Silva, 2015] da Silva, A. R. (2015). Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155.

[Dumas et al., 2013] Dumas, M., Rosa, M. L., Mendling, J., and Reijers, H. A. (2013). *Fundamentals of Business Process Management.* Springer, Berlin Heidelberg.

[Eclipse Foundation, a] Eclipse Foundation. Eclipse Modeling Framework (EMF). `http://www.eclipse.org/modeling/emf/`.

[Eclipse Foundation, b] Eclipse Foundation. Graphical modeling framework (gmf). `http://www.eclipse.org/modeling/gmf/`.

[Eclipse Foundation, c] Eclipse Foundation. Xtext - language development framework. `http://www.eclipse.org/Xtext/`.

[Efftinge et al., 2008] Efftinge, S., Friese, P., Haase, A., et al. (2008). openArchitectureWare User Guide. `http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/index.html`.

[El Kouhen et al., 2012] El Kouhen, A., Dumoulin, C., Gerard, S., and Boulet, P. (2012). Evaluation of Modeling Tools Adaptation. Technical report.

[Erdweg et al., 2013] Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G. D. P., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., van der Vlist, K., Wachsmuth, G. H., and van der Woning, J. (2013). *The State of the Art in Language Workbenches*, pages 197–217. Springer International Publishing, Cham.

[Erdweg et al., 2015] Erdweg, S., van der Storm, T., Völter, M., Tratt, L., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G. D. P., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., van der Vlist, K., Wachsmuth, G., and van der Woning, J. (2015). Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47.

[Frank, 2014] Frank, U. (2014). Multi-level modeling - toward a new paradigm of conceptual modeling and information systems design. *Business & Information Systems Engineering (BISE)*, 6(3).

[Goldschmidt et al., 2012] Goldschmidt, T., Becker, S., and Burger, E. (2012). Towards a tool-oriented taxonomy of view-based modelling. In Sinz, E. J. and Schürr, A., editors, *Modellierung 2012*, pages 59–74, Bonn. Gesellschaft für Informatik e.V.

[Gronback, 2009] Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Longman, Amsterdam.

[Grönniger et al., 2014] Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2014). Textbased modeling. *CoRR*, abs/1409.6623.

[Gulden, 2009] Gulden, J. (2009). Minimal invasive generative Entwicklung von Modellierungswerkzeugen (minimal invasive generative development of modeling tools). In Fischer, S., Maehle, E., and Reischuk, R., editors, *Tagungsband der Konferenz INFORMATIK 2009, Lübeck, 28.9.2009 – 2.10.2009.*

[Gulden, 2016] Gulden, J. (2016). Recommendations for data visualizations based on gestalt patterns. In *Proceedings of the Enterprise System (ES) 2016 conference in Melbourne, 2016-11-02 – 2016-11-03.* IEEE.

[Gulden and Reijers, 2015] Gulden, J. and Reijers, H. A. (2015). Toward advanced visualization techniques for conceptual modeling. In Grabis, J. and Sandkuhl, K., editors, *Proceedings of the CAiSE Forum 2015 Stockholm, Sweden, June 8-12, 2015*, CEUR Workshop Proceedings. CEUR.

[Gulden et al., 2016] Gulden, J., van der Linden, D., and Aysolmaz, B. (2016). Requirements for research on visualizations in information systems engineering. In *Proceedings of the ENASE Conference 2016, April 27-28 2016, Rome.*

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274.

[Karsai et al., 2014] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkel, S. (2014). Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378.*

[Karsai et al., 2003] Karsai, G., Sztipanovits, J., Ledeczi, A., and Bapty, T. (2003). Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164.

[Kelly, 2013] Kelly, S. (2013). Empirical comparison of language workbenches. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, DSM '13, pages 33–38, New York, NY, USA. ACM.

[Kelly and Tolvanen, 2008] Kelly, S. and Tolvanen, J.-P. (2008). *Domain Specific Modeling: enabling full code-generation.* Wiley.

[Kern et al., 2011] Kern, H., Hummel, A., and Kühne, S. (2011). Towards a comparative analysis of meta-metamodels. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, pages 7–12, New York, NY, USA. ACM.

[Kolovos et al., 2017] Kolovos, D. S., García-Domínguez, A., Rose, L. M., and Paige, R. F. (2017). Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255.

[Krahn et al., 2010] Krahn, H., Rumpe, B., and Völkel, S. (2010). Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372.

[Mahemoff and Johnston, 1999] Mahemoff, M. J. and Johnston, L. J. (1999). Handling multiple domain objects with model-view-controller. In *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 32*, pages 28–39.

[Marquard et al., 2016] Marquard, M., Debois, S., Slaats, T., and Hildebrandt, T. (2016). Forms are declarative processes! In Rosa, M. L., Loos, P., and Pastor, O., editors, *Proceedings of the Business Process Management Forum 2016, Rio de Janeiro, Brazil, September 18-22, 2016*. Springer.

[Moody, 2009] Moody, D. L. (2009). The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779.

[Nijholt, 1988] Nijholt, A. (1988). *Computers and languages: theory and practice*. Studies in computer science and artificial intelligence. North-Holland.

[Obeo, 2020] Obeo (2020). Obeo designer. `https://www.obeodesigner.com/`.

[Odell, 1994] Odell, J. J. (1994). Power types. *Journal of Object Oriented Programming*, 7(2):8–12.

[OMG, 2015] OMG (2015). OMG Unified Modeling Language (OMG UML) version 2.5. `http://www.omg.org/spec/UML/2.5`.

[Paige et al., 2017] Paige, R. F., Kokaly, S., Cheng, B., Bordeleau, F., Storrle, H., Whittle, J., and Abrahao, S. (2017). User experience for model-driven engineering: Challenges and future directions. In *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*. Institute of Electrical and Electronics Engineers Inc.

[Pfeiffer and Pichler, 2008] Pfeiffer, M. and Pichler, J. (2008). A comparison of tool support for textual domain-specific languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, pages 1–7.

[Reenskaug, 2007] Reenskaug, T. (2007). The original MVC reports. Technical report, University of Oslo, Oslo.

[Rose et al., 2012] Rose, L. M., Kolovos, D. S., and Paige, R. F. (2012). Eugenia live: a flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 15–20. ACM.

[Scheer and Schneider, 2005] Scheer, A.-W. and Schneider, K. (2005). Aris – architecture of integrated information systems. In Bernus, P., Mertins, K., and Schmidt, G., editors, *Handbook on Architectures of Information Systems*, pages 605–623. Springer, Berlin, Heidelberg.

[Stahl and Voelter, 2006] Stahl, T. and Voelter, M. (2006). *Model-Driven Software Development – Technology, Engineering, Management*. Wiley, Chichester.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *Eclipse Modeling Framework*. Addison Wesley, Amsterdam, 2nd edition.

[Tolvanen and Kelly, 2009] Tolvanen, J.-P. and Kelly, S. (2009). Metaedit+: Defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 819–820, New York, NY, USA. ACM.

[Tolvanen and Rossi, 2003] Tolvanen, J.-P. and Rossi, M. (2003). Metaedit+: Defining and using domain-specific modeling languages and code generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 92–93, New York, NY, USA. ACM.

[Viyović et al., 2014] Viyović, V., Maksimović, M., and Perisić, B. (2014). Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238.

[Voelter, 2013] Voelter, M. (2013). *Language and IDE Modularization and Composition with MPS*, pages 383–430. Springer, Berlin, Heidelberg.

[Voelter et al., 2013] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., and Wachsmuth, G. (2013). *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org.

[Voelter and Lisson, 2014] Voelter, M. and Lisson, S. (2014). Supporting diverse notations in MPS' projectional editor. In *GEMOC@ MoDELS*, pages 7–16.

[Vujović et al., 2014] Vujović, V., Maksimović, M., and Perišić, B. (2014). Comparative analysis of DSM graphical editor frameworks: Graphiti vs. Sirius. In *23nd International Electrotechnical and Computer Science Conference ERK, Portorož, B*, pages 7–10.

[Weske, 2012] Weske, M. (2012). *Business Process Management: Concepts, Languages, Architectures*. Springer, Berlin Heidelberg, 2nd edition.

[White, 1969] White, A. T. A. (1969). *Critical Path Method*. Prentice Hall Press.