Domain Engineering for Software Tools

Tony Clark and Balbir S. Barn

Abstract General purpose software engineering tools are expensive to develop and maintain, and often difficult to learn and use. Domain-specific tools tend to be small, focussed and easier to learn; however, domain-specific tooling tends to be technology specific and therefore introduces interoperability problems. This chapter provides a contribution to DSL tool development by describing a *language-driven* approach to domain engineering whereby a tool is modelled in terms of the syntax and semantics of the language it supports. This chapter uses UML to define a simple class modelling language and its tooling.

Keywords Domain specific language • Meta-modelling • Software tools

1 Introduction

The proliferation of methods in the 1980s and early 1990s yielded to the overwhelming force brought about by the development of a unified language: UML and its variants. The adoption of a single language provided the economic incentive to produce supporting tools. A number of commercial tools emerged, but perhaps because of the general purpose nature of the language, it rapidly became apparent that development, support and marketing of such tools was no small task. Many tools dropped away leaving a small number of commercial players trying to support a very diverse market. The result is that tools to support UML-style development are like Swiss-Army Knives: they aim to support all possible tasks with little guidance.

Software tools and technologies fall into two broad categories: *horizontal* and *vertical*. A horizontal technology can be applied across a wide range of application

T. Clark (✉) · B.S. Barn
Middlesex University, London, UK
e-mail: t.n.clark@mdx.ac.uk; b.barn@mdx.ac.uk

areas because it provides general purpose functionality. A vertical technology is specific to a particular application domain. Perhaps as a reaction to the lowest-common-denominator (LCD) flavour of UML, there has been a rise in interest in Domain Specific Languages (DSL) and their associated tools. Unlike UML which is a horizontal technology, a DSL aims to be a vertical technology by providing a means to represent, analyse and manipulate concepts from a specific domain such as real-time systems, telecomms and transport networks.

DSL tools can afford to be simpler than the UML equivalents because they expose very specific functionality. However this simplicity comes at a price. Where UML aims to support all aspects of the software development process and therefore can guarantee interoperability, the DSL approach leads to tool chains that can have interoperability problems [6]. In addition, because UML adopts LCD, it is possible to use it in a wide range of diverse domains. Typically each software engineering project is unique and therefore a DSL approach leads to a new toolset for each project.

The problem to be addressed is how to gain the benefits of both approaches. UML represents a universally applicable, interoperable, cost-effective technology that suffers from complexity and insufficient support for project domains. While attempts to use the stereotyping capability within the UML standard are numerous, tool capability is limited. DSLs produce technologies that are simple and focussed but suffer from complexity of development and interoperability issues. We would like to be able to produce DSL tools *in a standard way* that is universally applicable, leads to tool interoperability and therefore makes the development of DSL tools cost effective without leading to huge one-size-fits-all platforms.

Our proposal is to apply domain engineering techniques at the tool level using a standard tool meta-language. A tool model expressed in the tool meta-language can take two forms: a *specification* of the required tool and a tool platform *configuration* model. The former contains a model of the domain and specifies tool functionality over the domain, the latter is consistent with the specification and can be uploaded into any (proprietary or open-source) suitable framework that will configure itself appropriately.

Domain Engineering of Tool Models, as envisioned by our proposal, would be a phase of any new software engineering project. Tool models from similar projects could be used as a starting point, and modelling techniques such as transformations, slicing and merging can be brought to bear. Tool specifications would be useful as a teaching aid, and model slicing could be used to produce minimal sub-sets whose functionality could be gradually increased to produce an incremental training programme.

Tool models must contain a variety of features in order to be effective. Each domain can be considered as a language consisting of syntax and semantics. The tool itself can also be considered as a language whose syntax involves menus, editors, tree-browsers, etc., and whose semantics involves events that cause changes to the application domain. A model will describe how the state of a tool is serialized in order to guarantee interoperability. Models should also contain non-functional aspects such as usability and efficiency.

Our claim is that it is possible to take a domain engineering approach to tool modelling. A tool modelling standard would require a collaborative effort involving experts from a wide variety of disciplines. This chapter validates the claim by providing an overview of a simple meta-language for specifying tool models, using the language to specify a simple tool, and finally analysing the model in terms of the resulting benefits.

This chapter is organized as follows: Sect. 2 describes related approaches to developing modelling tools; Sect. 3 describes our approach to applying language engineering to tool development; Sect. 4 defines a class modelling language using the proposed language-driven approach; Sect. 5 extends the language to produce a tool definition using the approach; Sect. 6 discussed the approach and outlines further work.

2 Related Work

Software engineering is still essentially a young discipline and while efforts to compare it with other engineering disciplines invariably result in criticism of the discipline, some of the characteristics of software engineering are really the result of progress in trying to address the challenges of intense design. Young and Faulk provide an account of how this intense design manifests itself [22]. For example, unlike in other engineering areas, SE includes considerable activity in designing the design processes (see Pedreira et al. for relatively recent systematic literature review on tailoring software processes[18]). Closely related to the design process is the use of abstractions and notational elements. While abstraction is an essential step in any field of endeavour, in SE, the choice of abstraction is a defining characteristic of a software design process and can have direct and practical consequence on the output of a software design process as noted in the pioneering work of Parnas [17]. Similarly the use of notations to support abstractions is also a dominant field of enquiry. Thus programming languages, requirements descriptions [12], test plans [3] and so on have all been studied from a notational perspective.

Collectively, these activities have coalesced into ubiquitous approaches to software development manifested as methods, concepts and their associated software tools. There is no space to enumerate each and every one but examples include: The Information Engineering Facility from Texas Instruments [11], Software through Pictures [20] and of course the unification of modelling concepts, and notations through the Unified Modeling Language (UML) and its leading proponent for many years, Rational Rose and subsequent versions. As the use of UML grew, the complexity of the availability of concepts, notations, and the rules of their usage became much more of challenge and methods such as the Rational Unified Process developed alongside UML to help manage the complexity.

Despite the expressive power of UML, activity, consistent with SE research for designing the design processes, proceeded to extend UML or create variants of UML to support particular niche needs. For example, Egyed and Kruchten modified UML and the Rational Rose toolset to support Architecture Modelling [7]. While such extensions are popular, they are technology oriented and do not present themselves at sufficient level of abstraction for end-users. As Kelly notes: “Simply put, UML is not domain-specific, and UML tools were not designed to support changing UML. Trying to build domain-specific models in a UML tool is—at best—like trying to write English in a Spanish version of Word” [13]. However, this style of modification that capitalized on the unique power of the conceptual structures when modifications are applied to the structures themselves remains a key strategic method in dealing with complex design requirements for producing software and so we can observe “abstractions of abstractions” and even the design of processes for design processes. Such a reflective approach or meta-engineering to modification creates a unique view of software engineering that is distinct to engineering in general as it is not constrained by material processes or laws of physics.

The idea of organizations designing processes to suit their software development practice and to have these supported by toolsets led to the development of meta-modelling toolsets. Although the original meta CASE (Computer aided Software Engineering) tools such as Ipsys Tool Builder [1], have long gone there are still two key toolsets widely used currently. These are MetaEdit+ [14] and GMF.[1] Egbert and Hainaut describe four necessary components that require modelisation in order to support the generation of tools from a meta-engineering environment [8]: (1) meta model (ontology and repository); (2) their interface (specification, representation and control); (3) their functions/processes (transformations and evolutions) and (4) their methodologies (the reasoning and activity guidelines). Both MetaEdit+ and GMF provide modelisation of these components. But it could be argued that the proprietary nature of the repository associated with MetaEdit+ prevents transformations to external tools and therefore sharing and evolution outside of the toolset. GMF and its dependence upon the Eclipse framework is simply too complex and so prevents users from designing domain-specific tools.

Tool frameworks emerged in recent years including Visual Studio and Eclipse. These frameworks allow the developer to tailor the platform to suit a particular domain by supplying static descriptions (often in XML) of the tool functionality and organization. However, they remain very platform specific, are incomplete, and require detailed implementation knowledge to work effectively.

A problem with tool descriptions is that they often use an implementation-specific or proprietary format. They often do not address semantics and where they aim to be self-contained, they present a mechanism for constructing a visual editor for a language such as MetaEdit+[2] and the Miarama toolset [9] (although see [4] that discusses semantic interoperability of DSMLs).

Language engineering can be argued to be a key feature of software system engineering and the underlying methods and technologies of meta-modelling [19] and domain-specific language engineering [21] have matured over the last decade to

[1] http://www.eclipse.org/modeling/gmf/.

[2] http://www.metacase.com.

the point where they can be applied in industrial situations. However, the application of language engineering to tool development has not seen significant research. This chapter represents a contribution to this field.

3 Language Engineering for Tools

Our proposal is that Software Engineering (SE) is a language-based discipline. The underlying design of software artefacts arising from an SE project should be located in the design of an appropriate language. Thus any SE project involves one or more domains that should be identified and precisely defined as languages. Furthermore, tools are required in order to engineer domain artefacts and tool definition should be part of the domain engineering process. Once a precise requirement for a DSL tool is available, a project is in a position to find the most appropriate way of providing the tools and their associated languages.

This approach can lead to a specification for a project-specific tool suite that supports domain-specific languages. As such it represents a lens through which all concrete technologies can be viewed in order to achieve consistency and clarity across a project. Taken further, the approach can lead to executable models in which case, given the availability of a suitable meta-tool platform, the project-specific tool suite can be automatically generated.

This chapter addresses the specification of languages and tools, and this section describes an approach for engineering these domains. Section 3.1 describes potential approaches to language description, Sect. 3.2 shows how one approach can be implemented using standard modelling techniques and Sect. 3.3 shows how the approach incorporates tooling.

3.1 Domains as Languages

Languages are the medium of communication between humans. Geographical distance can explain the difference between Chinese and English, but other than that we might initially be tempted to conclude that languages in geographical areas are essentially the same. However, there is a rich diversity of special purpose languages that have arisen, some of which cross-cut national boundaries, in order to support meaningful communication in specialized areas, or *domains*. Most professional disciplines constitute such domains, for example medicine (try interpreting your doctor's notes!) and programming (where bugs cannot fly but need to be exterminated). The importance of language and its relevance to computer system and organizational interaction has been researched at depth by foundational work [5,15].

Computer systems are controlled by making demands on their behaviour and requesting information about their state. The particular demands and requests for any application constitute a language of discourse with the system. Like languages associated with nation states, there is a general purpose language for controlling families of systems; in the early days of computing this was the mode of discourse. However, like discourse in professional domains, such low-level communication was seen to be error prone and verbose. This led to the development of high-level languages.

Software engineering differs from other engineering disciplines, such as Civil, Electrical and Chemical that are each based on a collection of fixed rules that are the same for each new system. Software is capable of defining new execution rules for each new system or family of systems through programming. The dichotomy of program and data is not fixed and allows programs to be represented as data, and data to be interpreted by programs. This recursive relationship can be extended as far as required in order to shield the engineer from the low-level communication medium. Therefore, what works for human-to-human communication (i.e., the development of domain specific languages of discourse) can be applied to software systems with the same benefits.

Given that software engineering can be viewed as a language-based activity, how frequently should a language be developed and who should do it? Once for all applications by hardware specialists? Once for each operating system? Once for each development style (OO, logic)? Once for each type of application (real-time, graphical, distributed)? Once for each application (booking system, share transaction system)? Once per use by users? Clearly there is a spectrum and it will depend on the range of variability required as to where the requirement to engineer and use languages lies. In general, one could argue that a general application user would not require the ability to specialize the medium of communication, although many systems allow a degree of control over system properties that can be viewed in this light.

Commercial software development is increasingly striving for application families, or *product lines*. Such an approach is attractive because it can abstract key commonalities and isolate variation points. In order to succeed with this approach it is necessary to be clear about what constitutes a language and what range of variability is available to the engineer. A *language* for software engineering must have a single definition that we will call a model of its *abstract syntax*. In software terms, the abstract syntax is a computer-friendly data representation of the language and provides the reference-point for all other definitions of the language; the exact format of the data definition is a variation point for language definition. Typically, humans find abstract syntax difficult to work with because it is verbose, therefore a language has one or more *concrete syntax* definitions. A concrete syntax for a language may take the form of strings of text or of graphical displays, or both [2]; it provides a variation point and is often a matter of taste. A concrete syntax definition may take the form of a grammar and there will be a relationship between the concrete and abstract syntaxes.

A language should have a *semantics*. A semantics is typically defined as a model of a *semantic domain* (separate from the abstract syntax model) and a relationship between the abstract syntax and the semantic domain. Degenerately, there is nothing special about the semantics of a language, it is composed from two models and a relationship and is defined for a particular purpose. A semantics may be defined in order to deduce properties of language elements, or to provide a mechanism for defining valid and invalid abstract syntax configurations. Often when people talk of *the* semantics of a language, they really mean one of the possible relationships that can usefully be defined.

Computer Science has defined a number of categories of language semantics [16], including:

Denotational. Where each language element is to be viewed as directly representing a semantic domain element. In this case the semantic definition takes the form of a predicate that holds between elements from each domain. For example, a language for class modelling has models that denote sets of objects and links. The objects are required to be well-formed instances of classes in the model and links are instances of associations that must hold between appropriate objects and must satisfy multiplicity constraints attached to the association ends.

Operational. Where each language element is to be viewed as representing a sequence of steps in a calculation. In this case the semantic definition can take the form of an algorithm that performs the steps, or the form of a relationship between a syntax element and a state sequence. For example, an interpreter can be defined that processes a state machine and a collection of externally generated events in order to control an object.

Axiomatic. Where each language element takes part in a relationship with semantic domain elements that represents facts that can be deduced and the semantics takes the form of a theory. For example, a class model may include operations with pre and post conditions written in OCL. From such a model, properties of legal execution traces can be deduced.

The particular semantic category that is chosen will depend on the type of language and its intended use. Two obvious cases exist in software engineering which make a distinction between *static* and *dynamic* systems. Typically a static system represents some information, such as a relational database or an XML document, expressed using a convenient domain-specific language. A dynamic system executes in some way, for example a business process that manages a collection of databases. Static systems lend themselves to denotational semantics and dynamic systems lend themselves to operational semantics. Both static and dynamic systems can be expressed using axiomatic semantics.

3.2 A Language Based Method

The previous section has presented the case for languages-based software engineering and has outlined the components necessary to define these kinds of languages. To use this approach on a particular project, a specific technology and language definition method must be selected. There are many technologies that are suitable for defining languages including BNF grammars, symbolic programming languages, DSL tools, graph grammars, term rewriting systems. For the purposes of this chapter we will use UML class diagrams together with invariant constraints expressed using OCL since these have been standardized and are widely used. In addition we will limit ourselves to static languages and denotational semantics.

UML is to be used to *specify* DSL tools. Therefore models are used to express the language features described in the previous section and predicates are used to define relationships between the language features. As such the models are required to express *what* needs to hold between the models, even when the models describe dynamic features such as the actions performed by a tool when a user selects a menu item. This is to be contrasted with DSL models that express *how* the relationships are constructed; such models would be executable and are not considered further.

A specification in UML can be expressed using *relationship* models. A relationship model consists of a collection of classes that hold between elements of two or more *domain* models. The domain models are to be viewed as being imported into the relationship model. Each class in the relationship model is linked to domain model classes using associations and OCL invariants on the relationship classes are used to specify when the relationships hold between domain instances. The choice of UML leads to the following:

Abstract syntax. A domain model showing the concepts and relationships in the language. OCL is used to define syntactic constraints between elements. A typical syntactic constraint would require all names to be unique in a given context.

Concrete syntax. A domain model for a collection of display elements. Typical modelling examples would be a graphical display language consisting of boxes, text and lines. In principle there may be more than one concrete syntax model. The case study in Sect. 4 uses graphical elements such as box and text because the language lends itself to a diagrammatic syntax. A general purpose concrete syntax domain may be extended, for example class-boxes can be defined as a collection of sub-boxes and text. Well-formedness constraints are expressed using OCL.

Syntax relationship. A relationship model between abstract and concrete syntax elements. This model defines the constraints on the legal concrete representations of well-formed language constructs.

Semantic domain. A domain model whose elements are to be used as the meaning of abstract syntax elements. It is not necessary to have the abstract syntax and semantics elements to be in one-to-one correspondence. OCL is used to express well-formedness constraints on configurations of semantic elements.

Semantic relationship. A relationship model between abstract syntax and semantic domain. It is important to note that there may be more than one semantic domain and therefore more than one semantic relationship. However, in practice it is usual to have a single semantics and therefore the relationship model can be viewed as defining *the* semantics of the language.

A language is rarely defined from scratch; it is usually based on existing languages. If it is a textual language, then there is generally a fairly standard expression sub-language, and if it is a graphical language, then it tends to be based

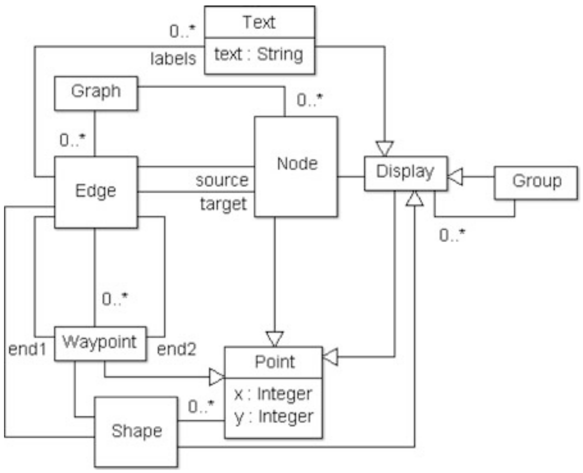


Fig. 1 General concrete syntax

on a graph or a tree. The language used in our case study is graphical and the concrete syntax will be based on the general purpose graphical syntax defined in Fig. 1. A diagram is a graph consisting of nodes with edges between them. Each node occurs at a point on the diagram and has a display element that is used to draw the node on the screen. A display element can be some text, a shape made up of points with lines between them, or is a group of displays. Each edge has a source and target node, and a collection of waypoints. An edge has a minimum of two waypoints that are the edge-ends. An edge has a collection of labels and each waypoint of an edge has an option shape used to decorate the waypoint.

3.3 The Tooling Domain

Our proposal is that tooling should be included in any domain-specific approach to software engineering. By providing a specification of the required tooling for a language, a project documents the intended usage of a DSL and clearly expresses the requirements for any concrete tools that are to be used. Where multiple languages are to be used, it is possible to test concrete tool frameworks against the combined tool requirement model.

Given this proposal, how are tools to be specified? This can be determined by analysis of the key features of any tool. A tool exists in a collection of states that can be changed by interaction with its environment; this is equivalent to the abstract syntax of the language as defined above. Since the tool must manage instances of a language, its states are linked to abstract syntax instances.

Most software tools have a user-interface that consists of tree browsers, text editors, property editors, graphical editors etc. The user-interface is equivalent to the concrete syntax of a language as described above and the same approach of a relationship model between abstract and concrete syntaxes can be used.

A tool performs tasks, often in response to user interaction through a collection of input gestures via buttons, menus etc. Tool execution can be modelled as a collection of state-transition traces, where each step in a trace contains before and after states, and an event. Such executions constitute the semantics of a tool, defined in terms of a semantic domain and a relationship model.

Therefore, a tool can be modelled in terms of the same features as a language as described in the previous section. If the tool is to be used to manage a language, then the two models can be linked via relationship models so that the language specification restricts legal tool behaviour and the tool restricts the features of the language that are available to the user.

This approach is attractive because it places no restrictions on how the tool is realized. The behaviour can be left ambiguous where further detail is not required. A typical ambiguity is diagram layout where any concrete tool would be free to use any algorithm. Multiple tools that are specified in this way can be combined using relationship models to produce a single unified tool model; each component tool will place restrictions on the functionality and behaviour of the other.

4 Case Study

The previous section has argued the case for a language-based approach to tool-based domain engineering. This section provides an overview of applying this approach to a domain. We have chosen a domain that is widely understood in order that the key steps of the process are clear. The case study uses class modelling as the domain; the following sections address each major step in the approach: the abstract syntax of class-based modelling is defined in Sect. 4.1; the language for *drawing* class models is defined in Sect. 4.2; Sect. 4.3 defines a relationship between the elements of the class models and the concrete syntax, the relationship specifies when a drawing is a legal representation of a model. For the case study we are taking a denotational approach to language engineering and therefore Sect. 4.4 defines a model of the semantic domain for class models: every class model denotes a snapshot contain objects and links. Given an abstract syntax and a semantic domain it is necessary to define when a semantics (i.e., a snapshot) is a well-formed instance of a class model, this is done by defining a relationship between elements of these two models in Sect. 4.5.

Note that this section aims to give the key features of all elements of the approach. As such some of the models omit elements that would otherwise be necessary to provide a complete definition where the inclusion of these elements simply repeats key features that are exemplified elsewhere. In addition it should be noted that association ends on models are labelled only when necessary and that the names of association ends default to the names of the attached classes with a lowercase initial letter and internalized capitalization. Multiplicities on association ends default to 1 unless otherwise indicated.

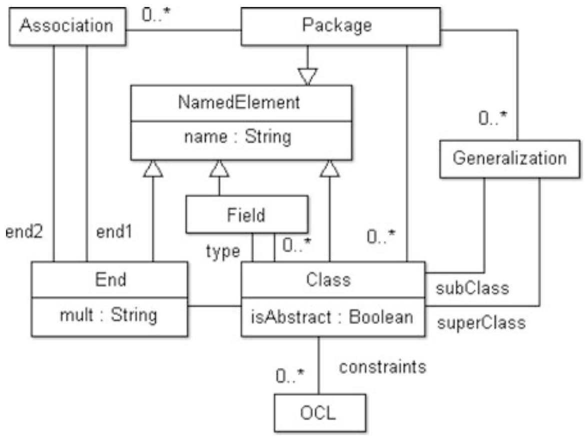


Fig. 2 Class models abstract syntax

4.1 Abstract Syntax Domain Model

The abstract syntax structure of class models is shown in Fig. 2. A package contains a collection of classes that can be related using associations and generalizations. Each class has a number of fields and a collection of constraints. Each constraint is an OCL expression; it is beyond the scope of this chapter to define OCL abstract syntax; however, the reader is assumed to be familiar with standard first-order predicate calculus which OCL approximates. An association has two ends attached to classes, each end has a name and a multiplicity. The syntax for class models has been chosen so that it is the basis for the languages used in the case study. A full and complete model for class modelling is of course available in the UML 2.x specification available at: http://www.omg.org/spec/UML/2.0/. Our presentation is necessarily a simpler version for the purposes of illustrating key concepts of our approach.

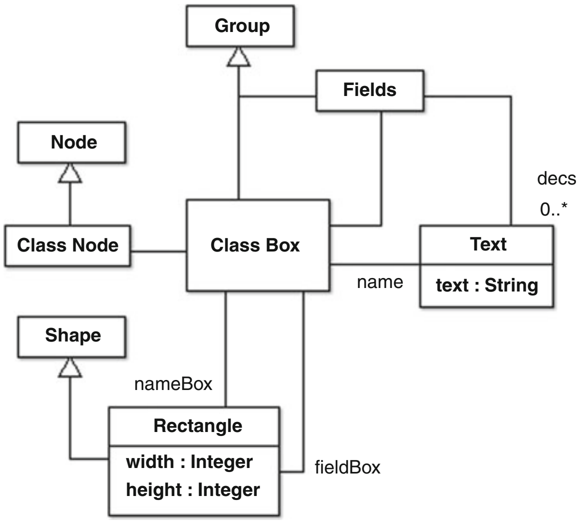
A domain has a collection of well-formedness constraints. it will not be possible with the scope of the chapter to define all of the well-formedness constraints for class diagram domain and relationship models. Therefore, we will give a representative sample. For example, the following constraints requires that all classes in a package have different names:

```
context Package inv:
  classes->forAll(c1 c2 | c1.name = c2.name implies c1 = c2)
```

4.2 Concrete Syntax

In general, the concrete syntax of a language will be built from some basic elements. The concrete syntax of class diagrams is defined as an extension to the domain model given in Fig. 1. The extensions specialize the basic concrete elements so that it is easy to define the mapping model between the abstract and concrete syntaxes.

Fig. 3 Class nodes



The specializations for classes are shown in Fig. 3 where the class `ClassNode` is a specialization of `Node` that represents classes. Classes are displayed as rectangles containing text, so `Shape` is specialized to produce `Rectangle`. `ClassBox` is a specialization of `Group` that is required to contain the display elements for a class name and its fields. OCL is used to require the elements to be correctly formed, for example the following is a fragment that forces the nested rectangles in a class box to be positioned correctly:

```
context ClassBox inv:
  displays = Set{nameBox,fieldBox,name,fields} and
  nameBox.x = x and nameBox.y = y and
  fieldBox.x = x and
  fieldBox.y = y + nameBox.height and
  nameBox.width = nameBox.width
```

The domain model in Fig. 4 defines a specialization of edges that can be used to represent associations and generalizations on a diagram. As before OCL can be used to constrain the domain, for example the type and positioning of association edge connections (where `contains` holds between a display and a point when the display contains the point):

```
context AssociationEdge inv:
  source.oclIsKindOf(ClassNode) and
  target.oclIsKindOf(ClassNode) and
  source.display.contains(end1) and
  source.display.contains(end2)
```

4.3 Syntax Mapping

A syntax mapping is a model that links elements from the abstract syntax domain to appropriate elements in the concrete syntax domain. In general the mapping

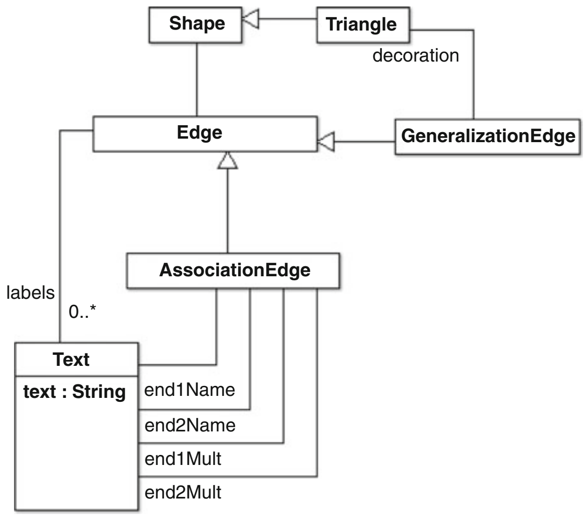


Fig. 4 Class model edges

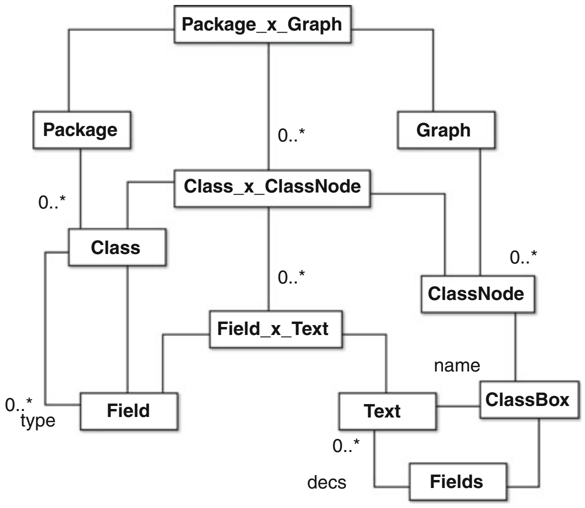


Fig. 5 Class syntax mapping

describes the minimum constraints necessary to ensure that the rules for constructing and displaying elements of the language are correct. Typically the syntax mapping will leave issues such as layout underspecified so that tools can implement their own algorithms. In addition, a syntax mapping may leave features such as colour and the use of icons unspecified where these are not important.

Figure 5 shows the mapping model for classes within packages. It shows a typical pattern that occurs in mapping models between two domains `X` and `Y` whereby mapping classes `A_x_B` are constructed for class `A` of domain `X` and class `B` of domain `Y`.

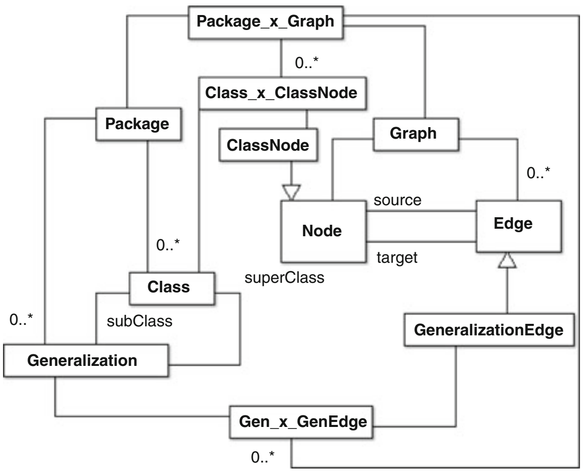


Fig. 6 Generalization syntax mapping

The root mapping class is `Package_x_Graph` that defines a relationship between a package and a graph. The OCL constraints attached to a mapping class define the conditions under which instances of the associated domain instances can be related. For example, a graph is a correctly formed package when it has a class node for each class in the package:

```
context Package_x_ClassNode inv:
  package.classes = classNode_x_classes.class and
  graph.nodes = classNode_x_classNodes.node
```

Each mapping class must have appropriate constraints. For example, the name of a class must be associated with the text in the class node of a class node:

```
context Class_x_ClassNode inv:
  class.name = classNode.classBox.name.text and
  class.fields = field_x_texts.field and
  classNode.classBox.fields.decs = field_x_texts.text
context Field_x_FieldText inv:
  text.text = field.name + ":" + field.type.name
```

The mapping model for generalizations between classes is shown in Fig. 6. The mapping constraints are not defined here, but are of a similar form to those defined for `Package_x_Graph`.

4.4 Semantic Domain

The semantic domain of a language defines the meaning of the elements from the abstract syntax domain. Class models defined in Fig. 2 are static since there is no way of defining dynamic features such as operations. Class models are well understood and therefore the semantic domain shown in Fig. 7 is perhaps obvious. However, when working with a new domain, the semantics might be less obvious. In these situations it is worth considering designing the semantic domain before the

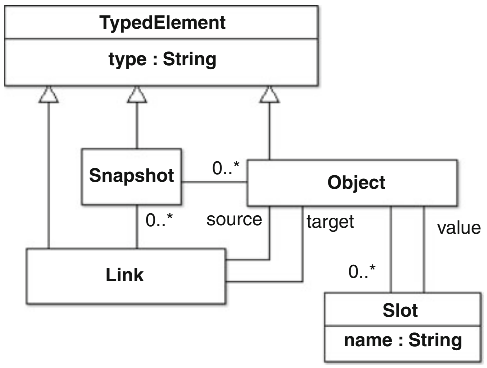


Fig. 7 Class semantic domain

syntax domains. This can be done by constructing a model of the things that are to be denoted and then designing the syntax domains in such a way as to provide a convenient way of denoting the semantic domain elements.

As with the syntax domains, a semantic domain includes OCL constraints that express well-formedness constraints. For example, snapshot constraints should include a requirement that object fields must have unique names and links can only hold between objects that are in the same snapshot.

4.5 Semantic Mapping

The semantic mapping associates abstract syntax elements with semantic domain elements. In the case of packages, classes are associated with objects so that the slots and fields match up as shown in Fig. 8. The following OCL constraint requires that packages have snapshots as instances when the objects in the snapshot are all instances of corresponding classes in the package:

```
context Package_x_Snapshot inv:
  package.classes->includeAll(snapshot.objects) and
  package.classes = class_x_objects.class and
  snapshot.objects = class_x_objects.object
```

Each instance of a class must have slots that correspond to the fields of the class:

```
context Class_x_Object inv:
  class.fields = field_x_slots.field and
  object.slots = field_x_slots.slot
```

The names of fields and slots must match up:

```
context Field_x_Slot inv:
  field.name = slot.name
```

The type of a field must correspond to the type of a slot value:

```
context Package_x_ClassNode inv:
  class_x_objects.field__x_slots->forAll(r1 |
    class_x_objects->exists(r2 |
      r1.class = r1.field.type and
      r1.object = r1.slot.value))
```

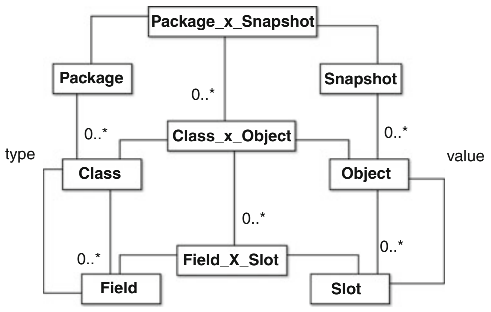


Fig. 8 Instance semantics

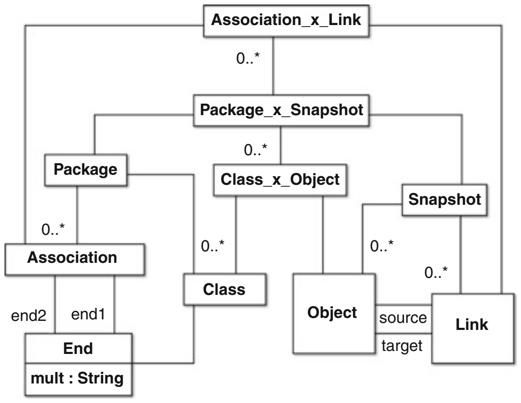


Fig. 9 Association semantics

If we assume that each OCL expression has a predicate `satisfiedBy` that returns a boolean value when supplied with an object then the invariants on classes are specified as:

```
context Class_x_Object inv:
  class.constraints->forAll(c | c.satisfiedBy(object))
```

The semantic mapping for associations and links is defined in Fig. 9. The multiplicity on association ends places a constraint on the number of links that can occur in the snapshot. For example, suppose that `A` is an association between classes `X` and `Y` with the multiplicity 1 on the end (`end1`) attached to `X`. In a snapshot we equate a link `source` with `end1` and a link `target` with `end2`. Therefore, for a link to be a valid instance of `A`, the source must be a valid instance of `X`, the target must be a valid instance of `Y` and, for each instance of `Y` there can be at most one instance of `A`. The multiplicity constraint is expressed in OCL as follows:

```
context Package_x_Snapshot inv:
  package.associations->forAll(a |
    a.end1.mult = "1" implies
      snapshot.objects->forAll(o |
        association_x_links->select(r |
          r.association = a and
          r.link.target = o)->size >= 1
```

We omit the constraints relating to generalizations that require an object to have slots for all the super-classes of a class.

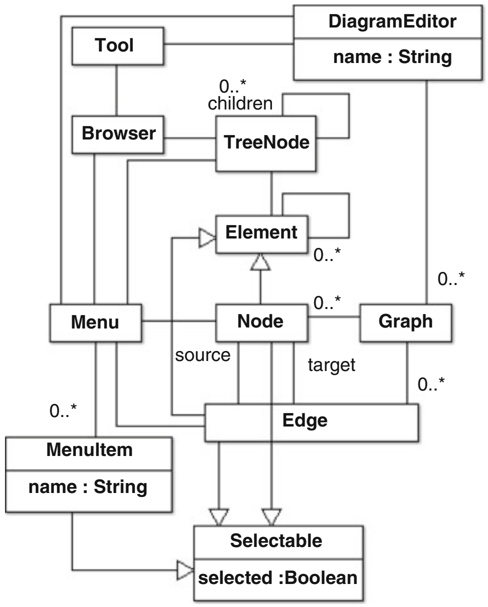


Fig. 10 Tool abstract syntax

5 Tooling

Software tools are a huge investment in terms of development and maintenance. Many tools tend to be large and complex to learn. The ability to specify the requirements of a tool for a specific task makes it clear how any suitable tool should be used and also makes the required functionality independent of any specific technology platform. Once defined, a tool can be reused by transforming its specification to a new language. This section provides an outline on how a tool can be defined for the class modelling domain by using a language-based approach.

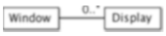
5.1 Abstract Syntax

Figure 10 defines an abstract syntax domain for a simple general-purpose modelling tool. The tool consists of a browser and a diagram editor. The browser contains tree structured data and the diagram editor contains a graph. Various elements of the tool have menus associated with them.

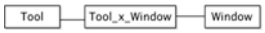
Note that the tool model is abstract in the sense that it does not specify the format of the tree structured data or the graph data. Therefore, OCL can be used to place some requirements on the tool model, for example that menu items are all unique in a menu, that will apply to any tool that is specified to be consistent with the model via a mapping model.

5.2 Concrete Syntax and Syntax Mapping

The concrete syntax of a tool is defined using a model of the required graphical elements to be used for browser nodes, menus and diagrams. This can simply use the class `Display` defined earlier:



The abstract and concrete syntax of a tool are combined using a mapping model:



The mapping class `Tool_x_Window` has constraints that require the window to include suitable arranged display elements. For example, the tree nodes of a browser node must be displayed as text elements with positions that are arranged as a tree, and the edges of a diagram must be displayed as lines whose end positions match the locations of the attached nodes. It is beyond the scope of the current chapter to give the details of the `Tool_x_Window` mapping class; however, it should be noted that the approach allows the mapping to be underspecified with respect to the exact display elements used for tool features such as menus, diagram nodes and waypoint decorations. In addition, the mapping may also be used to allow certain usability features of the required tools to be specified. For example, if colours are available in the `Window` concrete syntax model, then certain mixtures of colours can be defined as illegal.

5.3 Semantics

In general, a software tool is a reactive system that performs actions in response to a stimulus provided by the user. Each action causes the tool to change state and then wait for the next stimulus. Figure 11 shows a simple semantic domain for a tool as a *filmstrip* consisting of ordered steps that perform tool actions. A simple tool for class modelling will require actions that create, delete and move elements. More sophisticated tools will include actions for drag-and-drop, save and load, user input etc.

5.4 Semantic Mapping

Our proposition is that a tool can be treated as a domain-specific language and therefore should be specified using a semantic mapping. The semantic mapping

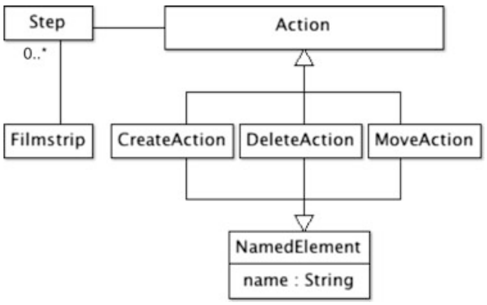


Fig. 11 Tool semantics

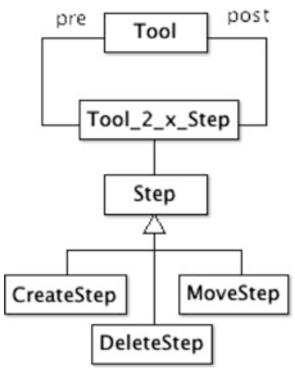


Fig. 12 Tool step semantic mapping

will associate elements of the tool abstract syntax with the tool semantic domain. In the case of simple class modelling, the semantic mapping must define what happens for the creation, deletion and movement actions. Figure 12 defines a mapping model consisting of a pre and post tool state and a step. The constraints attached to the mapping class must be carefully constructed in order to specify the required state changes, although some of the details may be left under-specified in order to allow tools the greatest amount of freedom in satisfying the specification. For example, the following constraint requires that a browser node is created in response to a create action. It assumes that a tool provides the query operation `hasSelectedBrowserNode()` which is true when exactly one browser node is selected and `getSelectedBrowserNode()` that gets the selected node. In addition we assume that we can perform `p - q` for two tools `p` and `q` that will produce the elements in `p` that are not elements in `q`.

```
context Tool2_x_CreateStep inv:
  pre.hasSelectedBrowserNode() and
  post.hasSlectedBrowserNode() and
  let n = pre.getSelectedbrowserNode()
      n' = post.getSelectedBrowserNode()
  in n.menuItems->exists(i | i.name = step.action.name) and
     (post - pre) = n'.children - n.children and
     n'.children->size = n.children->size + 1
```

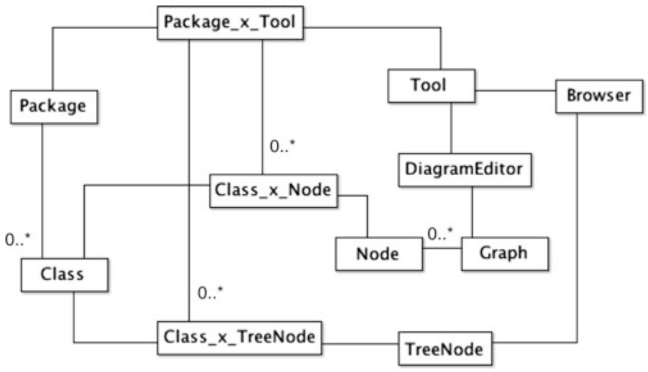


Fig. 13 Mapping abstract syntax

5.5 Mapping Languages and Tools

A tool is used to manage elements in a language. Therefore, the abstract syntax of the language must be associated with the abstract syntax of the tool. In the case of class modelling, packages and package elements are associated with the appropriate tool elements such as browser nodes and diagram nodes. Figure 13 shows a mapping model that defines the appropriate associations for classes (associations and generalizations are omitted). OCL constraints are used to tie the elements together, for example:

```
context Package_x_Tool inv:
  package.classes = class_x_nodes.class and
  tool.nodes = class_x_nodes.node and
  package.classes = class_x_treeNodes.class and
  tool.browser.topLevelTreeNodes() = class_x_treeNodes.treeNode
```

Having linked the abstract syntax of the tool and class modelling language, the concrete syntax of both languages must be mapped onto each other. Although verbose, this is a straightforward mapping model that involves point-wise constraints between the display elements used to construct class models and the display elements that are drawn on a window. The mapping class `P_x_G_x_T_x_W` associates the mappings `Package_x_Graph` and `Tool_x_Window`:

Package_x_Graph
P_x_G_x_T_x_W
Tool_x_Window

the following OCL constraint shows how the two mappings are associated:

```
context P_x_G_x_T_x_W inv:
  t_x_w.displays->includesAll(p_x_g.graph.nodes.display) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.shapes) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.waypoints.shape) and
  t_x_w.displays->includesAll(p_x_g.graph.edges.waypoints.labels) and
  t_x_w.displays->select(t | t.oclIsKindOf(Text))->
    includesAll(p_x_g.package.classes.name) and
```

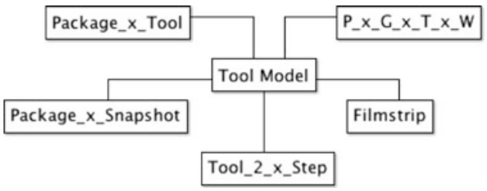


Fig. 14 The complete tool specification

```
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.name) and
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.end1.name) and
t_x_w.displays->select(t | t.oclIsKindOf(Text))->
  includesAll(p_x_g.package.associations.end2.name)
```

The mapping defined above requires the tool window to contain all the appropriate display elements but does not place any restrictions on where the information is displayed. The tool syntactic mappings defined above require that the information is appropriately displayed via browser areas and diagram areas.

The tool for the case study in this chapter has been defined using a language-based approach in terms of its abstract syntax, concrete syntax and its semantic mapping. As such it is sufficiently underspecified and may be used to specify a tool for any language that involves a browser and diagram elements. We have then defined point-wise mappings between class model syntax and tool syntax and between class model semantics and tool semantics. It remains to ensure that all of the point-wise mappings hold simultaneously, thereby completing the tool specification. Figure 14 shows the mapping `Tool Model` that maps all the component mappings. The associated OCL constraint (not shown) on `Tool Model` requires that the appropriate elements of each constituent mapping tie up correctly. Therefore, both syntactically and semantically the tool will behave as required for all steps in a filmstrip.

6 Analysis and Conclusion

This chapter has identified a problem with software engineering tools and modelling tools in particular that the investment required to produce the tools leads to large complex general purpose platforms that are expensive to use and difficult to learn. A solution to this problem is to take a domain-specific approach to tooling, thereby producing lean focussed tools that are appropriate to each new project. Whilst domain-specific tools solve some problems, they introduce others since they are often based on proprietary technologies that lead to interoperability issues.

Our proposal is to take a domain engineering approach to tooling so that the tools required for a project are specified as models and are then mapped on to existing general purpose technologies or on to domain-specific technologies as appropriate. We have described a method for specifying domain-specific tools based on UML and used the method to specify a simple tool for class modelling.

The benefits of domain-specific tool models include a precise description of tools and the languages they support. The models can be mapped to the functionality of existing platforms and the tool semantics can be used to check that the platforms satisfy the required semantics. In addition tool models can be reused. For example, the class modelling tool defined in this chapter can be extended to include features such as components and state machines. Furthermore, the tool models can be sliced to restrict their functionality, for example a class modelling tool that does not support generalization. Since the tool model contains both the language and its tool functionality, any extension or restriction of the language must also indicate the extra tool functionality or the tool functionality that can be hidden from the user.

The method and case study in this chapter has shown that it is possible to specify a tool in terms of a domain and mapping models. The case study that has been used is an example of a *horizontal* domain and it remains to show that the approach works for a variety of vertical domains. In addition the horizontal domain, although familiar, is self-referential in the sense that a class-modelling tool has been specified using class models. It remains to apply the approach to another horizontal domain that is not used in the definition of itself.

Our goal is to be able to construct executable models of tools and to provide a standard meta-language for tools that can be consumed by a wide range of tooling platforms. A number of approaches to this problem have been tried, but most do not explicitly model all of the features of a model. A successful approach, that unfortunately uses a non-standard representation but otherwise shows that the approach could be generally applied, is the meta-modelling tool platform XMF that is compared against similar software tools in [10].

References

1. Alderson, A.: Meta-case technology. In: Software Development Environments and CASE Technology. Lecture Notes in Computer Science, vol. 509, pp. 81–91. Springer, Berlin (1991)
2. Andrés, F., de Lara, J., Guerra, E.: Domain specific languages with graphical and textual views. In: AGTIVE, pp. 82–97, 2007
3. Bauer, J.A., Finger, A.B: Test plan generation using formal grammars. In: Proceedings of the 4th International Conference on Software Engineering (ICSE'79), pp. 425–432. IEEE, Piscataway (1979)
4. Chiprianov, V., Kermarrec, Y., Rouvrais, S.: Meta-tools for software language engineering: a flexible collaborative modeling language for efficient telecommunications service design. In: FlexiTools: Workshop on Flexible Modeling Tools at the 32nd ACM/IEEE ICSE Intl. Conf. on Software Engineering, 2010
5. Clancey, W.J.: Understanding computers and cognition: A new foundation for design. Artif. Intell. **31**(2), 232–250 (1987)
6. Clark, T., Bettin, J.: Editorial for the theme issue on model-based interoperability. Software Syst. Model. **11**(1), 7–10 (2012)
7. Egyed, A., Kruchten, P.B. Rose/architect: a tool to visualize architecture. In Proceedings of the Thirty-Second Annual Hawaii International Conference on System Sciences (HICSS'99), vol. 8, p. 8066. IEEE Computer Society, Washington, DC (1999)

8. Englebert, V., Hainaut, J.L.: Db-main: a next generation meta-case. Inform. Syst. **24**(2), 99–112 (1999)
9. Grundy, J.C., Hosking, J.G., Huh, J., Na-Liu Li, K.: Marama: an eclipse meta-toolset for generating multi-view environments. In: ICSE, pp. 819–822, 2008
10. Helsen, S., Ryman, A.G., Spinellis, D.: Where's my jetpack? IEEE Software **25**(5), 18–21 (2008)
11. Texas Instruments Incorporated: IEF: Methodology Overview, 2nd edn. Texas Instruments, Plano (1990)
12. Jacobson, I.: Object-oriented software engineering: a use case driven approach. Pearson Education India, 1992
13. Kelly, S.: Comparison of eclipse EMF/GEF and metaedit+ for DSM. In: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development, 2004
14. Kelly, S., Lyytinen, K., Rossi, M.: Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In: Advanced Information Systems Engineering, pp. 1–21. Springer, New York (1996)
15. Kensing, F., Winograd, T.: The language/action approach to the design of computer-support for cooperative work: A preliminary study in work mapping, Number 27. Stanford University. Center for the Study of Language and Information, 1991
16. Milne, R., Strachey, C.: A Theory of Programming Language Semantics, 99th edn. Halsted Press, New York (1977)
17. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Comm. ACM **15**(12), 1053–1058 (1972)
18. Pedreira, O., Piattini, M., Luaces, M.R., Brisaboa, N.R.: A systematic review of software process tailoring. ACM SIGSOFT Software Eng. Note **32**(3), 1–6 (2007)
19. Sprinkle, J., Rumpe, B., Vangheluwe, H., Karsai, G.: Metamodelling - state of the art and research challenges. In: Model-Based Engineering of Embedded Real-Time Systems, pp. 57–76, 2007
20. Wasserman, A.I., Pircher, P.A.: A graphical, extensible integrated environment for software development. In: ACM Sigplan Notices, vol. 22, pp. 131–142. ACM, New York (1987)
21. Weisemöller, I., Schürr, A.: A comparison of standard compliant ways to define domain specific languages. In: MoDELS Workshops, pp. 47–58, 2007
22. Young, M., Faulk, S.: Sharing what we know about software engineering. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, pp. 439–442. ACM, New York (2010)