# Rapid Design of Meta Models

Bastian Roth, Matthias Jahn, and Stefan Jablonski
Chair for Databases & Information Systems
University of Bayreuth
Bayreuth, Germany
{bastian.roth, matthias.jahn, stefan.jablonski} @ uni-bayreuth.de

*Abstract* - **Designing concise meta models manually is a complex task. Hence, newly proposed approaches were developed, which follow the idea of inferring meta models from given model examples. Unlike most approaches in the state of the art, we accept arbitrary model examples independent of a concrete syntax. The contained entity instances may have assigned values to imaginary attributes (i.e., attributes that are not declared yet). Based on these entity instances and the possessed assignments, a meta model is derived in a direct way. However, this meta model is quite bloated with redundant information. To increase its quality, we provide recommendations for applying so-called language patterns like inheritance or enumerations. For this reason, the applicability of those patterns is analyzed concerning the available information gathered from the underlying model examples. In addition to our previously published work, we also support the derivation of meta model changes based on modifications and extensions of the initial example models. Furthermore, change recommendations are provided wherever possible. This new approach for iteratively building, modifying and refining meta models enables users to focus on the real world instances. Consequently, they are not distracted by keeping the meta level in mind and thus are able to design meta models rapidly.**

***Keywords - meta model derivation; meta model inference; derivation of meta model changes; refinement of meta models; language patterns***

## I. INTRODUCTION

In [1], we presented an approach how a concise meta model can be derived from a given set of example models.

The main aim of our work is to support users in defining domain specific languages (DSLs). In general, a DSL consists of three important parts: an abstract syntax, a concrete syntax and a set of semantic rules [2]. The abstract syntax defines language concepts and how they can be linked together. The concrete syntax in turn describes a notation for the visualization of the DSL, whereas the rule set defines the semantics of concepts of the abstract syntax.

Nowadays, developers of a DSL often tend to describe the abstract or concrete syntax with meta models [3]. These meta models are models that specify how their (instance) models are structured. Creating a meta model and hence a DSL is not a trivial task, if it has to be done manually. That is why different methods for developing meta models have been discovered. The most recent approach is the derivation of meta models out of some (possibly merely one) example models.

In the following, when talking from a meta model we always mean the abstract syntax of a DSL. Since it requires a large set of models, we explicitly do not support inference of constraint (e.g., based on OCL). Additionally to that, negative example models are needed as well to avoid overgeneralization [4], [5]. Negative examples are models, which expose an invalid scenario in terms of the intended DSL. In our case, providing such examples is impracticable because it forces the user to pre-think models that are out of the regarding domain's scope.

During the derivation of an abstract syntax, all meta model artefacts are generated automatically and thus, could differ from the user's expectations, especially in terms of quality. In order to achieve a tolerable degree of quality, the user is pointed to parts of the meta model with potential of improvement and also supplied with possible solutions in form of language patterns (e.g., inheritance or enumerations). In contrast to design patterns [6], language patterns are supported by modelling systems themselves and can be utilized in a direct and simple manner.

The development of a meta model is often driven by the evolution in understanding of the domain of interest. Hence, together with the growing knowledge, the meta model often needs to be adapted to fulfil the domain's requirements. Therefore, it is essential that – based on modifications of the example models – changes within the meta model can be derived that define how such a meta model have to or may be adapted to get a concise result again. We call this whole process of incrementally deriving a meta model and providing some recommendations for quality improvements "rapid design of meta models".

After this introduction, some fundamentals are explained, which help understanding the later parts of this paper. Then, an example model is presented that is used for exemplary explanations through the entire paper. Following this, we introduce a method how a meta model can be automatically derived from a given set of such example models. Since this meta model may have some potential for improvements, in the subsequent Section V, two algorithms are presented that detect constellations of meta model elements with the aforesaid improvement potential. Also, each algorithm suggests a suitable solution, which can be applied by the user manually. Beyond tweaking the meta model, example models may be evolved as well or new ones can be added. Thus, in Section VI we describe an approach how freely performed changes at example model side have impacts on an already existing meta model. Afterwards, an overview of some related

work is given. Finally, we look out on future challenges in the field of rapid design of meta models and even whole domain-specific languages.

## II. FUNDAMENTALS

In the following three subsections, we explain some fundamentals that act as basis for the subsequently presented rapid design approach.

### A. Model Workbench

Model Workbench [7] is a web-based meta modeling platform that targets on supporting developers for creating their own modeling language. In contrast to other tools, it leverages advanced language patterns (e.g., Powertypes [8]) building (meta) models. Its implementation is based on the Orthogonal Classification [9]. Thus, the system provides a Linguistic Meta Model (LMM) [10] and interprets (meta) models at runtime in order to emulate a concrete textual syntax (called Linguistic Meta Language, LML). Together with that, Model Workbench is not limited to any number of meta levels since it is able to manage arbitrary meta model hierarchies. Therefore, it uses Clabjects [11] as a hybrid of a class and an object for representing concepts of a model (the term "concept" means a Clabject throughout the context of Model Workbench). Hence, a concept always has two different facets: a type and an instance facet. As a type (also called a meta concept), a concept defines attributes whereas as an instance (also called an instance concept), a concept contains assignments each of which may be associated with an attribute of an instantiated meta concept.

In general, Model Workbench divides attributes and assignments into two different classes depending on their respective type: literal and referential ones. Literal attributes can have one of the following types: boolean, integer, float, pointer, string or enumeration. In our understanding, enumerations are regarded as literal types, too. That is tolerable because enumerations can also be represented by integers with a highly restricted range of values. Each defined concept, however, may be used as a referential type.

### B. Modelling modes

Creating instance models based on a given meta model is a typical use case during modelling. Thereby, the instance models have to satisfy the constraints specified by the meta model. We call this kind of modelling the "stringent (modelling) mode".

By way of contrast, in context of the "free (modelling) mode" the constraints of a possibly available meta model are completely ignored. Accordingly, the LMM as specified in [12] needs to be expanded by schemalessness. Concretely, it means to be able to name an instance concept's type that does not exist (yet). Additionally, it must be possible to create assignments to imaginary attributes. An imaginary attribute is an attribute that is not (yet) declared by a meta concept.

### C. Essential assumption on equally named elements

The most important assumption we take is that equally named elements (types of instance concepts on the one hand, assignments and attributes on the other hand) always relate to the same semantic object at domain side. One could imagine a meta model containing two different concepts, each with exactly one string attribute labeled as `owner`. When trying to make this meta model more concise, both concepts are deemed to be candidates for generating a common super concept because of the two equally named attributes.

This assumption is mandatory. Otherwise, neither a meta model can be derived from one or more example models nor elements can be identified that exhibit some potential for improvement. Furthermore, the three inference approaches presented in Section VII follow a comparable principal.

## III. EXAMPLE MODEL

Before introducing the different algorithms for deriving a meta model, a linguistic example model (Figure 1) is presented on which we refer to in the following sections. This model is created freely using the LML as concrete syntax (i.e., there is no underlying meta model) and represents a process for planning a conference attendance. It only serves demonstration purposes and hence, it does not lay claim to

```
Start S
  next = Search

ElectronicTask Search
  title = "Conference search"
  next = Request
  duration = 1

PaperTask Request
  title = "Travel request"
  next = Split
  duration = 0.5

And Split
  next = Register, Booking

ElectronicTask Register
  title = "Conference registration"
  duration = 0.3
  next = Join

ElectronicTask Booking
  title = "Book Hotel"
  next = Organize

ElectronicTask Organize
  title = "Schedule journey"
  duration = 0.6
  next = Join

And Join
  next = E

DetachedTask Inform
  title = "Inform about topics"
  duration = 1

Exit E
```

Figure 1. Example model of a process

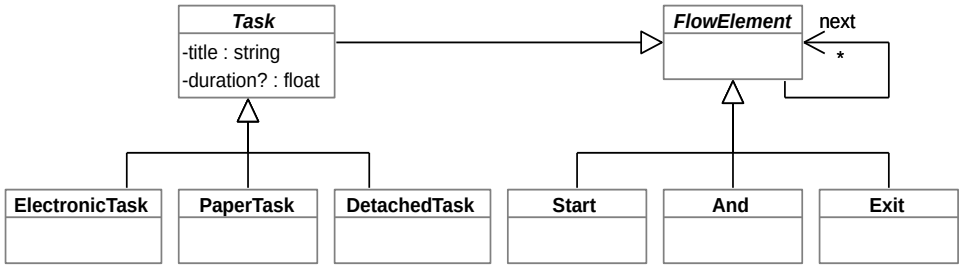


Figure 2. Meta model which matches exmple process model

contentual completeness. Since the according syntax (LML) is quite similar to the one of popular object-oriented programming languages, it is easy to read for software developers and modelers.

The process's flow is as follows. After a suitable conference has been found, an appropriate travel request needs to be submitted. Only then, a hotel may be booked and the journey may be scheduled. In parallel to these two steps, the researcher can also register at the conference. At any time, the scientist may inform herself/himself of the concrete topics covered during the conference.

The successor relationships are reflected in the `next` assignments. Furthermore, each task contains a `title` and can be equipped with a `duration`. The individual steps differ in that they have to be executed electronically (`ElectronicTask`), on paper (`PaperTask`) or besides at an undetermined time (`DetachedTask`). For the mentioned parallel processing, there are the two elements `Split` and `Join` with "and" semantic. The `And` means that all steps of both threads have to be completed before the execution can continue. Finally, there are two further elements, which determine the process's start and end points.

A meta model that matches this example process model is shown in Figure 2. It fulfills important quality criteria specified by Bertoa and Vallecillo in [13]. Looking at `ElectronicTask`, `PaperTask`, `DetachedTask`, `Start`, `And` and `Exit`, it exactly contains those concepts that are used within the example model (completeness). The same is true for the three attributes `title`, `duration` and `next`, which are declared only once and thus, redundancy is avoided. Moreover, because of the base concepts' naming – `Task` and `FlowElement` – their intention is obvious (self-documentation).

The meta model, however, concedes more flexibility as expressed by the underlying example model. For instance, `DetachedTask` is fully unconnected from the whole control flow, but the meta model states that it is a flow element nevertheless. The advantage of this additional flexibility is that when processing detached tasks, in some cases they need not be handled separately. For example, think about a concrete graphical syntax, which should be defined for this meta model. Then, it suffices if one containment mapping is specified for flow elements to lie within a certain process.

Suchlike assumptions concerning a higher degree of flexibility cannot be inferred from the example model. They require a profound knowledge about the particular domain and how according models are processed. Consequently, the meta model cannot be generated automatically as depicted by Figure 2, but it can be approximated to a certain degree. However, for further refinements, recommendations can be provided, which hint the user at sets of model elements with room for improvement. With improvements, we mean language patterns that can be applied to those model elements. Further details about this topic can be found in Section V.

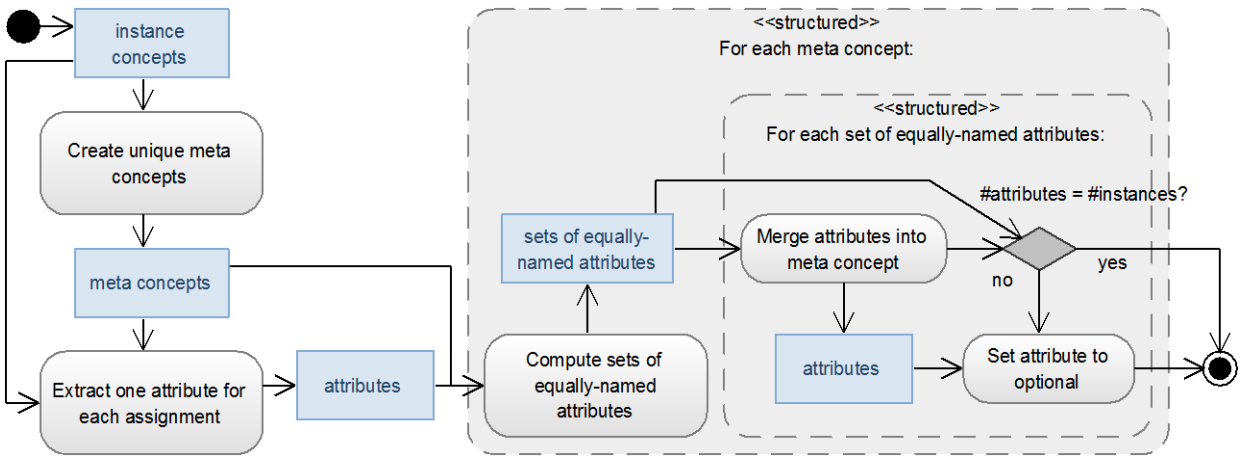


Figure 3. Activity diagram of the initial bottom-up algorithm

TABLE I. DERIVED META CONCEPTS WITH RESPECITVE ATTRIBUTE SETS

| Meta concepts | Instance concepts | Attributes | Attribute sets |
|---|---|---|---|
| Start | S | next: ElectronicTask | { next: ElectronicTask } |
| ElectronicTask | Search | title: string<br>duration: integer<br>next: PaperTask | { title: string, title: string, title: string, title: string }<br><br>{ duration: integer, duration: float, duration: float }<br><br>{ next: PaperTask, next: And, next: ElectronicTask, next: And } |
| | Register | title: string<br>duration: float<br>next: And | |
| | Booking | title: string<br>next: ElectronicTask | |
| | Organize | title: string<br>duration: float<br>next: And | |
| PaperTask | Request | title: string<br>duration: float<br>next: And | { title: string }<br>{ duration: float }<br>{ next: And } |
| And | Split | next[]: ElectronicTask | { next[]: ElectronicTask, next: Exit } |
| | Join | next: Exit | |
| DetachedTask | Inform | title: string<br>duration: integer | { title: string }<br>{ duration: integer } |
| Exit | E | | |

## IV. DERIVING AN INITIAL META MODEL

In the following, a method (Figure 3) is presented how a meta model can be derived from a set of example models. This method is an extension of the algorithm introduced in [1]. It exhibits some commonalities with the technique described in [14], but goes deeper into potentially occurring problems as well as respective solutions.

The algorithm's input are all instance concepts of the example models. At first, for each unique type name a separate meta concept is generated. Afterwards, for each assignment an associated attribute is created without allocating it to one of the previously generated concepts. Hereby, the upper bound of the attribute's multiplicity can already be determined. It is set to 1 if only one value is assigned, otherwise it is set to *. Identifying the attribute's type is done using regular expressions. For values that have one of the literal types boolean, integer, float, pointer (represented by qualified names) or string, the result is always unambiguous. However, in case only a qualified name is given, a further differentiation is required because the value may either represent another instance concept or an unspecified pointer. If an instance can be found whose name matches the assigned value, then the attribute type is set to the meta-concept of this instance. Otherwise, the attribute is declared as a pointer attribute.

After that, for every meta concept, sets of equally-named attributes are computed that act as base for the actual attribute declaration within the particular meta concept. Which attribute belongs to which meta concept can be ascertained by considering the underlying instance concepts.

For the example shown in Figure 1, TABLE I lists the derived meta concepts as well as the associated sets of equally-named attributes. In respect of a better traceability, the table also contains the underlying instance concepts together with the attributes inferred from the respective assignments. After the computation of the attribute sets, all attributes of each set are merged to one single attribute, which then is added to the particular meta concept. Merging attributes is not a trivial operation. Hence, it is explicated in the next subsection in more detail.

Finally, the last step checks whether the number of attributes of the original set is equal to the number of instances of the particular meta concept. If so, the algorithm terminates. Elsewise, the number of attributes is smaller than the number of instance concepts, which results in denoting the attribute as optional.

### A. Merging attributes

Merging attributes is the central activity when deriving a meta model because in doing so, the information and constraints stemming from different attributes are combined to one single attribute. This way, the domain knowledge obtained from the model examples is consolidated by considering the attribute's name, type and multiplicity. Since all attributes of the source set have the same name, it is adopted by the resulting attribute.

TABLE II. SUPPORTED LITERAL DATA TYPES WITH CONVERSION EXAMPLES

| | Boolean | Integer | Float | Pointer | String |
|---|---|---|---|---|---|
| **Boolean** | false / true | 0 / 1 | 0 / 1 | - | "false" / "true" |
| **Integer** | - | 3 / -2 | 3 / -2 | - | "3" / "-2" |
| **Float** | - | - | 0.5 / -3e6 | - | "0.5" / "-3e6" |
| **Pointer** | - | - | | X1 / A.B.c | "X1" / "A.B.c" |

*1) Merging attributes' multiplicities*

During the merging step, for multiplicities merely two values need to be regarded, namely 1 and 1..*. The multiplicity of an initially created attribute is set to 1 if the underlying assignments embraces exactly one value. In case of several values, the multiplicity is set to 1..*. Thus, when merging attributes only the multiplicity's upper bound can be determined. Thereby, the maximum value range is adopted (i.e., 1..* is preferred). Applied to the example from TABLE I, it means that the attribute set `next` of meta concept `And` leads to the multiplicity 1..*.

The lower bound is addressed in a downstream step. It only is set to 0 if there are more instances of the currently processed meta concept than attributes in the momentarily handled attribute set (see the decision node's successor in Figure 3). Then, instances exist, which do not possess an assignment to the current attribute. As an example, take a look at the `duration` attribute of `ElectronicTask` in TABLE I because it merely appears in three out of four instances.

*2) Merging attributes' types*

Conflating the types of attributes is far more complex. Thereby, literal and referential attributes need to be distinguished.

Literal attributes as defined by the LMM are attributes with one of these types: boolean, integer, float, string or pointer. In case two or more attributes with different literal types are detected, an automatic type conversion takes place, which is similar to the one of dynamic programming languages like JavaScript [15]. Thereby, the type with largest value range is adopted. Consequently, assigned values from a smaller value range have to be converted into the taken data type.

The head row of TABLE II lists all literal data types whereas the value range grows from left to right. Moreover, the table contains some conversion examples (from small to large value ranges). The type pointer occupies a special position in the context of an automatic type conversion since a pointer can solely be transformed into a string. Compatibility to other data types is not given, which results in aborting the derivation algorithm if such a scenario arises.

In TABLE I, a type conversion is required for the `duration` attribute of `ElectronicTask` since it is two times declared as float and one time as integer. Because of a larger value range the resulting type will be float.

If two or more attributes to merge feature different meta concepts as their type, for typing of the consolidated attribute, a common meta concept has to be determined as well. This use case is called Liskov substitution principle and is characteristically for the language pattern "generalization" / "inheritance" [16]. In case a common base concept already is available, it is set as the attribute's type. Otherwise, a suchlike base concept needs to be introduced first.

Referred to TABLE I, this affects the attribute sets `next` of `ElectronicTask` and `And`. As a consequence, for `ElectronicTask`, `PaperTask` and `And` as well as for `ElectronicTask` and `Exit` a base concept has to be created respectively. In Figure 4, these base concepts are represented as `ElectronicTaskOrPaperTaskOrAnd` and `ElectronicTaskOrExit`. The automatic naming happens by means of concatenating the names of the individual source concepts, whereas between two names always "Or" is inserted. Since the diagram shows the initially derived meta model for the example process model from Figure 1 all contained attribute sets are already merged and added to the respective meta concept. The question mark behind a literal attribute's name tells it is as an optional one (e.g., `duration`).

*B. Elimination of multiple inheritance*

As obvious through Figure 4, the approach presented above may lead to the introduction of multiple inheritance. In several cases this is undesired because it carries some potential risks [17] (e.g., name collision). That is why, an additional operation can be connected in series with the initial derivation process that removes multiple inheritance from the generated meta model. In order to not increasing the meta model's complexity artificially, multiple inheritance is replaced by the language pattern "single inheritance".

The replacement strategy starts by looking for compounds of concepts with multiple inheritance. For each such compound, all base concepts are identified and conflated to one common base concept using evolution techniques as described in [18].

The naming is handled equally to the one from above, i.e., names are concatenated using a connecting "Or". In order to restrict the name's length a bit, common partial strings are only quoted once.

Figure 5 depicts the accordingly modified variant of the meta model from Figure 4. The compound with multiple inheritance initially consists of `ElectronicTask`, `Exit`, `PaperTask`, `And`, `ElectronicTaskOrExit` and

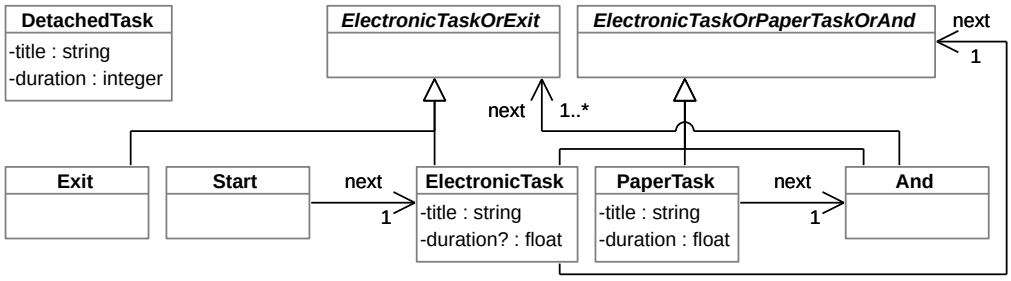


Figure 4. Initially derived meta model with multiple inheritance

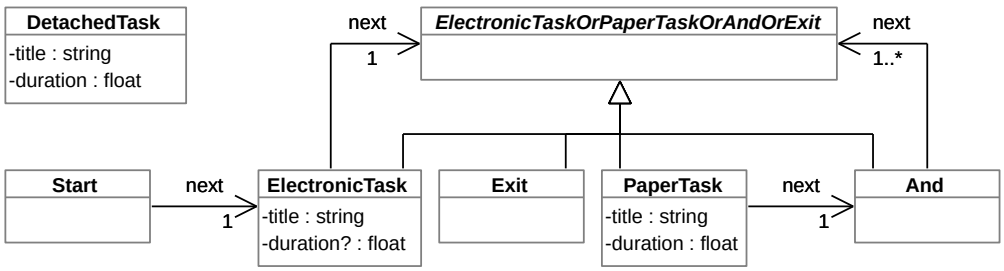


Figure 5. Initially derived meta model with single inheritance

`ElectronicTaskOrPaperTaskOrAnd`. The two latter mentioned meta concepts represent the base types, which are merged into `ElectronicTaskOrPaperTaskOrAndOr-Exit`.

Owing to later manual modifications, base concepts could also contain some attributes that again may result in naming conflicts. Such attributes have to be merged analogous to the method described in Section IV.A. Since this may lead again to more than one base concept per meta concept, the newly introduced multiple inheritance needs to be eliminated in turn. At the latest, this cycle terminates when a global base concept is found, which acts as generalization for all other meta concepts.

As an alternative to the foregoing strategy, instead of conflating the base concepts, the generalization hierarchy can be extended by introducing a super concept for those base concepts. If the concept compound comprises a big number of base concepts, it may result in a complex generalization hierarchy. Because of the large amount of additional concepts, the comprehensibility and thus the meta model's quality suffers [13]. However, the complexity of the meta model is only increased slightly when pursuing the first mentioned solution. Consequently, this one is preferred.

## V. META MODEL REFINEMENT

Looking at the initially derived meta model in Figure 5, some parallels to the expected variant in Figure 2 are indeed obvious, but the automatically generated model contains a bunch of redundancies, which impair its comprehensibility. Furthermore, the expected variant comprises already amended domain knowledge, which lacks the generated result. One example is the concept `Task` that specifies as a generalization which kind of information all tasks must/may provide. In this concrete case, it is about a task's title and a time designation how long a `Task` instance will take approximately.

Hence, the requirement arises to rebuild the derived result in a way that it widely corresponds to the expected model. Since inferred meta models can be much bigger than the ones shown in this article, it is desirable to point a modelling expert to constellations of model elements with potential for optimization. This is contrary to the method presented in [1] where optimizations are performed automatically by applying appropriate language patterns. The reason for limiting to recommendations comes from the amount of different possible solutions how a meta model may look like to fit a set of example models.

This becomes clear when looking at Figure 2, Figure 5 and Figure 6, which all are valid according to the example process model and only utilize single inheritance as language pattern. Which one to choose requires additional domain knowledge that is not available to the derivation engine. However, this knowledge is availble to the user and hence, (s)he can decide herself/himself whether to introduce a certain suggested pattern. Also, focusing on this challenge, we develop a framework that provides support for user-oriented meta model evolution [19].

To provide recommendations, we resort to the principle of equally-named attributes explicated in Section II.B. Thereby, in a given meta model, sets of concepts are searched, which declare as many equally-named attributes as possible. Suchlike sets represent candidates for introducing generalizing language patterns. The most widespread generalization pattern is single inheritance. It is addressed in the first subsection.

Another kind of generalization can be achieved using enumerations. An enumeration, however, does not relate to concepts but to literal data types with a limited value range. The basis are again equally-named attributes. This pattern is covered within the second subsection.

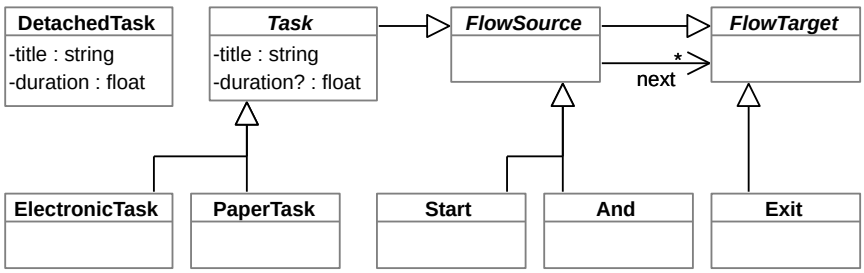


Figure 6. Alternative meta model with single inheritance

### A. Single inheritance as refinement recommendation

In order to provide a refinement recommendation for applying single inheritance, attributes need to be searched, which (potentially) have the same meaning. In the following, we call those attributes "corresponding attributes". With regard to Section II.B, two attributes correspond if they coincide by name and kind (i.e., referential or literal). By means of an external configuration it can be specified whether type and multiplicity also have to match such that correspondence is on hand. As opposed to equally-named attributes, corresponding attributes are declared by different meta concepts.

The described correspondence is an equivalence relation, because it is *reflexive* (each attributes corresponds to itself), *symmetric* (if attribute a corresponds to attribute b then b corresponds to a, too), and *transitive* (if attribute a corresponds to attribute b and attribute b corresponds to c then a corresponds to c as well). Consequently, the order of corresponding attributes is irrelevant and thus, it is expedient to represent them in form of sets.

Referred to the meta model in Figure 5, the following attribute sets arise as a result if besides the attribute names no further information is checked on equality:

- `{ DetachedTask.title: string, ElectronicTask.title: string, PaperTask.title: string }`
- `{ DetachedTask.duration: float, ElectronicTask.duration?: boolean, PaperTask.duration: boolean }`
- `{ Start.next: ElectronicTask, ElectronicTask.next[]: ElectronicTask-OrPaperTaskOrAndOrExit, PaperTask.next: And, And.next: ElectronicTaskOrPaperTaskOrAndOrExit }`

In case multiplicity is considered as well, the particular representatives of `ElectronicTask` of the `duration` and `next` attribute sets are dropped. For it, the `duration` is declared as optional while for the other concepts, it is specified as mandatory. The electronic task's `next` attribute, however, permits to assign multiple values whereas the other concepts require exactly one successor to be assigned.

A set of corresponding attributes implies that the declaring concepts of the attributes contained by this set exhibit exactly one correspondence, namely these attributes. In case of the first listed set, the three `title` attributes form a correspondence (communality) of the concepts `DetachedTask`, `ElectronicTask` and `PaperTask`. The same is true for the three `duration` attributes. Consequently, the three concepts `DetachedTask`, `ElectronicTask` and `PaperTask` possess exactly two communalities, which are determined by the two sets of corresponding attributes.

The issue of attribute sets with the same correspondences can be generalized. If two sets of corresponding attributes have the same size and the declaring concepts of the contained

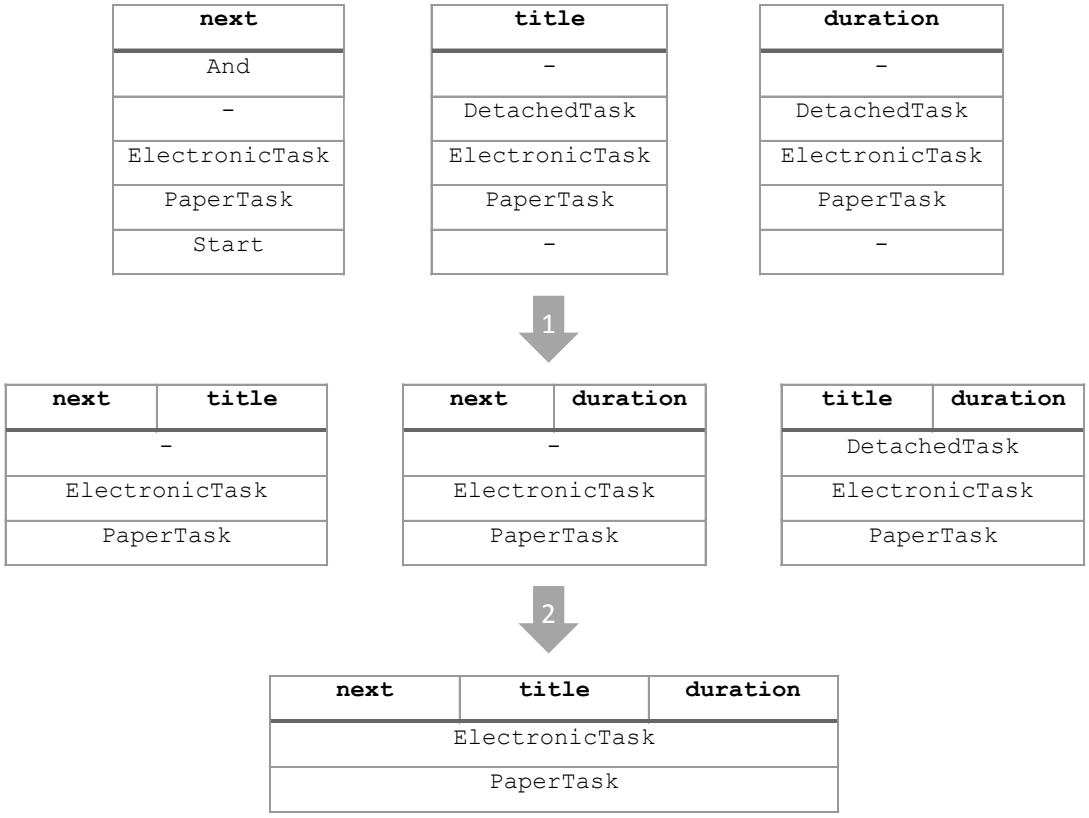


Figure 7. Example for determining dependent sets of corresponding attributes

attributes coincide, then we talk about a dependency between these attribute sets regarding the common parent concepts. Analogous to the corresponding attributes, this dependency relationship constitutes an equivalence relation.

Visualizing this circumstance can be done using tables like in Figure 7. Thereby, for each set of corresponding attributes, the first row contains exactly one entry with the common name of the respectively contained attributes. The rows below list all those concepts that depend on each other based on the attribute sets consolidated within the first row. Cells showing a "-" are included due to purpose of illustration without any contentual meaning. Each one of these tables represents a candidate for applying a generalizing language pattern and thus for refining the meta model in relation to the concepts and attributes listed by the table.

At large, in a meta model many such candidates can be found. Hence, it is important to weight the determined candidates and recommend them to the user ordered by this weight. It is defined by the number of dependent sets of corresponding attributes. As a consequence, a candidate is better than another one if there is a greater number of such attribute sets. In case this number is identical for two attribute sets, the quantity of declaring concepts is considered as secondary factor. It is justifiable because an in fact occurring communality is more probable if two or more concepts overlap in as many points (corresponding attributes) as possible. In Figure 7, it is the case for the table at the bottom. This table states that the concepts `ElectronicTask` and `PaperTask` depend on each other concerning the attribute sets `next`, `title` and `duration`.

The algorithm for determining all refinement candidates is shown in Figure 8 in form of an activity diagram. It starts with looking for corresponding attributes in a given collection of meta concepts. The specific correspondence criteria are predefined externally by means of an configuration.

The found sets of corresponding attributes are then converted into a data structure called "dependency tuple". Its content is exemplarily depicted by the tables in Figure 7. The first entry of such a tuple contains the dependent sets of corresponding attributes and thus, it conforms to the first rows of the example's tables. The second entry comprises those concepts, which declare exactly one attribute of every set of the first entry. These concepts are located in the other rows of the example tables. The three sets of corresponding attributes listed above are equivalent to the first three dependency tuples (represented as tables) in Figure 7.

The next step creates the initial dependency tuples and puts it at the beginning of the results list. The results list contains the refinement candidates, which are identified during the execution of the algorithm. The tuples are ordered descending by the quantity of included concepts. Accordingly, the first entry is always the candidate with the greatest probability in terms of an in fact occurring communality within the real world.

If there are at least two dependency tuples, they are combinated in pairs with formation of intersecting the declaring concepts. Thereby, dependency tuples are created only for such intersections, which contain at least two concepts since elsewise no dependency exists. This combination step is repeated as long as one tuple is left at a max. After that, the algorithm terminates and returns a list of refinement candidates ordered by the weight described above.

Applied to the example depicted by Figure 7, the results list looks as follows (for reason of clarity, solely the names of the corresponding attributes are specified):

```
({next, title, duration},
 {title, duration},
 {next, title},
 {next, duration},
 {next},
 {title},
 {duration})
```

At first place, it recommends the user to introduce a common base concept for `PaperTask` and `ElectronicTask`, which declares the three corresponding attributes `next`, `title` and `duration`. If (s)he does not want to do that (s)he can look at the next candidate. Based on the attributes `title` and `duration`, it recommends to introduce a base concept for `DetachedTask`, `PaperTask` and `ElectronicTask`. This can be continued until the last dependency tuple is arrived that only rests on `duration`.

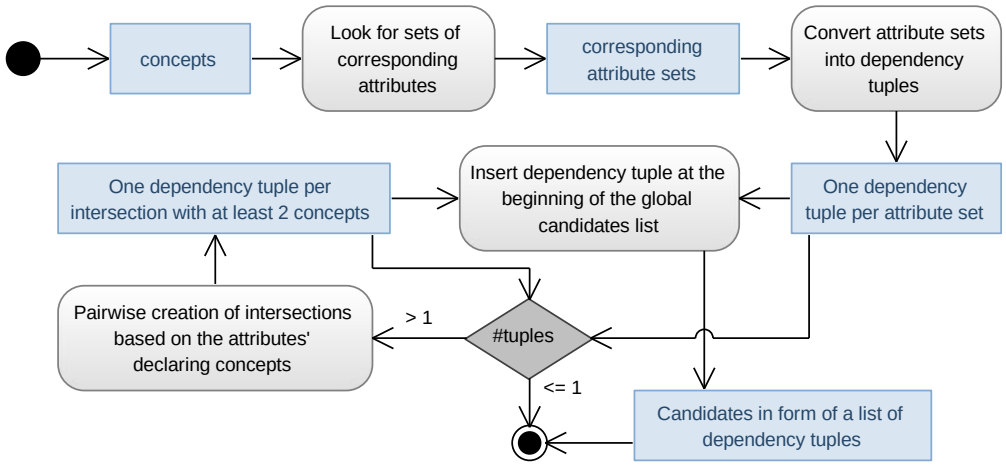


Figure 8. Algorithm for the computation of candidates to apply single inheritance

```
Job J1
  phase = PRE

Job J2
  phase = POST

Job J3
  phase = PRE

Job J4
  phase = DEFAULT

Job J5
  phase = DEFAULT

Job J6
  phase = PRE
```

Figure 9. Example model that induces a recommendation for introducing an enumeration

### B. Enumeration as refinement recommendation

An enumeration represents a data type with a strongly limited value range [20]. In general, it only consists of a few literals, which come into question as values for assignments. Therefore, recommending the introduction of an enumeration as data type is merely reasonable for corresponding attributes whose assignments exhibit repeatedly the same values. Owing to the equal lexical structure of pointers and enumeration literals, an enumeration can be only intended for pointer attributes by users. Consequently, in the current context merely two attributes may correspond to each other if they feature the same name and are of type pointer.

Additionally, the number of the different values should be stinted. However, a fix definition of where the border of "stinted" is exceeded cannot be given because this depends on the particular operational scenarios as well as the user's preferences. Instead, the analysis's focus lies on the repeated assignment of the same pointer values to corresponding attributes. Hence, it will be recommended to introduce an enumeration if at least two different values are repeatedly assigned to the same set of corresponding attributes. An example for a valid scenario is shown by Figure 9. It represents a model with six instances that all contain an assignment to the imaginary attribute `phase`. The derived meta model only consists of the concept `Job`, which manifests the aforementioned attribute `phase`. Since it is a pointer attribute and the literal `PRE` as well as the literal `DEFAULT` are used by at least two associated assignments (namely `J1`, `J3`, `J5` and `J4`, `J5`, respectively), a hint is generated that suggests to introduce an enumeration.

## VI. Deriving Meta Model Changes

When deriving meta model changes, the fundamental principle is to keep those changes to a minimum. Thus, the existing meta model only gets adapted insofar that modified or newly added example models become valid. This is necessary because users are allowed to commute meta models arbitrarily. In case an existing meta model is always discarded and a complete re-generation takes place, all manually performed modifications would be lost. Which modifications are performed at the meta model automatically during the repeated derivation is explicated in the first subsection.

In the second subsection, we seize the idea of recommendations. Primarily, these recommendations can be seen as counterparts to the explicit and implicit impacts on the meta model presented in Section IV and Section V.

### A. Required changes

In order to ensure the conformity of the example models with regards to the meta model, in any case those artefacts of the models need to be extracted that conflict with the meta model. Potential for conflicts is carried by the LMM's parts, which are extended about schemalessness (Section II.B). On the one hand, these are type names of concepts and on the other hand these are names of assignments.

If free modelling mode is enabled, the user may equip new instance concepts with a type name of a not yet available type (meta concept). Suchlike instances are handled the same way as during the initial inference of a meta model (Section IV). A user may also change a type name of an existing instance concept, which is already linked with a meta concept, such that it does not fit with any other available meta concept. Then, this concept is considered as new, too. Furthermore, potentially present assignments are broken away from their underlying attributes. Afterwards, processing can continue in the same way as with completely new instances.

The free mode enabled, new dynamic assignments (i.e., assignments without an underlying attribute) can be created inside of instance concepts, which already have an associated meta concept. For every suchlike assignment an appropriate attribute is generated, but without putting it into a meta concept. After that, per meta concept sets of equally-named attributes are determined. Each of these sets is merged to one attribute and added to a particular meta concept, according to the method described in Section IV.A. Thereby, an existing meta concept is expanded by an attribute that matches one or more dynamic assignments.

Furthermore, assignments with an underlying attribute may feature arbitrary values on the right side provided that the respective intention (referential or literal) is not violated. Assuming that there is an integer attribute with name "height", then assignments may be of any other literal type in free mode. For instance, a meaningful value would be 3.5 although it is outside the value range of integers. Deriving the according meta model changes would convert the "height" attribute's type to float. Here again, strategies are reused, which have been introduced for inferring an initial meta model (Section IV). In case of an underlying literal attribute, a type conversion occurs towards a larger value range (examples are depicted by TABLE II). For referential attributes, however, a common base concept is required, which has to be created if not yet existent.

In addition, enumeration attributes need to be handled separately. Valid values are basically pointers that do not reference another concept. If values are specified without a suitable enumeration literal, an according literal is generated and added to the enumeration. Beyond pointers, string values

can also be assigned to enumeration attributes while free modelling. This, however, leads to converting the underlying attribute to a string attribute. Besides, all enumeration literals are converted to strings as well. All other data types are not permitted and will result in aborting the derivation process in case they are used.

The presented five cases encompass all possible modification kinds of instance models that require a subsequent adaption of the underlying meta model to achieve validity when modelling stringently.

### B. Change recommendations

The different types of change requirements can be divided into three categories. In the first category there are all recommendations that affect the value ranges of attributes. The second category encompasses recommendations to delete certain concepts of the meta model. The third one contains those recommendations that refer to a removal of language patterns. Therewith this class stands inverse to the suggestions presented in Section V.

#### 1) Narrowing of attribute contraints

During the derivation of attributes all restriction in the model are softened. This is desirable for reasons of manual adaption. Instead, under certain circumstances narrowing the attribute's multiplicity or type can be recommended. For recommending the narrowing of an attribute's multiplicity, the minimum and maximum have to be handle separately. If all instances that can define an assignment do have such an assignment, a change of the minimum from 0 to 1 is suggested. Furthermore, narrowing the maximum of the multiplicity can be useful if the current value is * and all assignments are just single valued.

Dealing with the attribute's type requires again to distinguish between literal and referential attributes. A literal attribute can be checked whether all according assignments have a lower range than previously defined (TABLE II). In this case a replacement of the old type with the new literal type can be recommended. One could imagine that an attribute's type is float and all assigned values are within the integer rage. Hence, a change of the attribute type may be expedient.

Referential attributes can be handled in a similar way. However, they are tested whether a generalization of their type can be replaced by a specialization of it. Thereby, all assigned values have to be checked again. An example would be an attribute of type `ProcessOrAnd`. This type has been chosen because until now only processes and AND gateways have been assigned. During the next derivation of changes it is detected that only instances of `Process` were used as values. According to that, changing the attribute's type to `Process` is recommended.

#### 2) Concept removal

Based on changed instance models, a sure decision whether a concept is not needed any longer and thus can be deleted is hard to make. Every meta concept may be used in a model repository out of the current scope or needed within a code-generation step. That is why deleting a meta concept is not done automatically but could be done by a modelling expert who is supplied with a recommendation of an according deletion operation. A typical representative would be a non-abstract meta concept, which is not instantiated. Such a concept is a candidate for removal.

#### 3) Revocation of single inheritance

As stated above, the next case can be seen as opposite to the introduction of language patterns explicated in Section V. However, it claims for removing meta concepts, which again may lead to invalid external references. Hence, a model expert has to decide whether (s)he wants to adapt the model or not. If an abstract concept has exactly one specialization this concept is often obsolete. Thus, every concept that fulfils this constraint is a candidate for inlining into its specialization. One could imagine that the concept `PaperTask` (Figure 6) was removed manually. After that, a hint will be generated recommending the move of the two attributes `title` and

```
Start S
  next = Search

ElectronicTask Search
  title = "Conference search"
  next = Request
  duration = 1

PaperTask Request
  title = "Travel request"
  next = Split
  duration = "0.5"
  waitForReturn = true

And Split
  next = Register, Booking

ElectronicTask Register
  title = "Conference registration"
  duration = 0.3
  next = Join

ElectronicTask Booking
  title = "Book Hotel"
  next = Organize

ElectronicTask Organize
  title = "Schedule journey"
  duration = 0.6
  next = Join

And Join
  next = E

FreeTask Inform
  title = "Inform about topics"
  duration = 1

Exit E
```

Figure 10. Manually modified example model

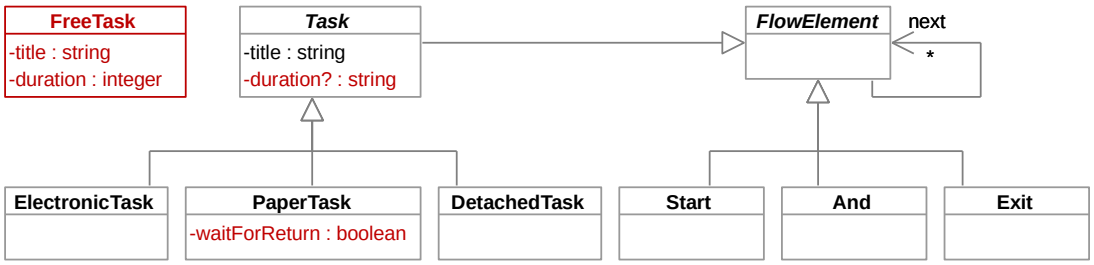


Figure 11. Automatically adapted meta model according to the modified example model

`duration` from `Task` to the specialization `Electronic-Task` and replacing the concept `Task` by `ElectronicTask` afterwards.

### C. Example

The example model depicted by Figure 10 largely represents the same process as shown by Figure 1. Both models only vary on three user-performed changes, which are highlighted in Figure 10 using red color. For the original example, a meta model has already been generated. Also, it has been adapted by the user so that it corresponds to the variant from Figure 2. Now, this meta model constitutes the foundation for deriving changes based on the modifications of the example model described below.

The first change concerns the `Request` concept's `duration` assignment that is already bound to a float attribute. The assignment of the floating point number `0.5` is replaced by the string `"0.5"` (fourth issue in Section VI.A). During the incremental meta model derivation, the type of the existing literal attribute is widened to string (concept `Task` in Figure 11) and all previously assigned values to this attribute are converted to their string representation. For instance, the value `1` from `Search` concept's `duration` assignment becomes `"1"`.

Directly below the modified duration assignment, a new assignment (`waitForReturn`) has been added without an underlying attribute (third issue in Section VI.A). Since it affects the one and only instance of `PaperTask`, this meta concept is simply extended by an appropriate Boolean attribute.

The third manipulation of the example model affects the `Inform` concept. Its type has been changed from `DetachedTask` to `FreeTask`, whereas no corresponding meta concept exists for the latter. According to the second issue in Section VI.A, a new meta concept called "FreeTask" is induced that also receives two attributes `title` and `duration`. As a result, there is no more concept, which instantiates `DetachedTask`. For that reason, the system suggests to the user to delete this concept (as per Section VI.B.2).

## VII. Related Work

As mentioned in the introduction, deriving a meta model from a set of model examples is not a totally new approach. Depending on their purpose, the available related work can be classified into two categories: meta model reconstruction and meta model creation.

Meta model reconstruction stems from the field of grammar reconstruction and grammatical inference [21]. Thereby, many textual sentences (ideally positive and negative samples) are analyzed to infer a grammar [22].

In current research, the Metamodel Recovery System (MARS) is one prominent representative for meta model reconstruction [5], [23]. It receives a set of model samples and transforms them to a representation that can be used by a grammar inference engine. The output of this engine (a grammar) is then converted back to an equivalent meta model. As the title suggests, MARS focuses on the recovery of meta models (e.g., if a meta model got lost). To obtain a meta model, which corresponds as much as possible to the original one, a large number of positive model samples is required. Otherwise, the resulting meta model is strongly restricted in its capabilities. Since we mostly receive only one or at least a small set of model examples this approach is not practicable for us.

Up to our knowledge, there are three research groups that generate a meta model by deriving it from very few model examples. BitKit as one representative has a rather different intention [24]. Its authors aim at supporting the pre-requirements analysis of software products by allowing to model in a freeform way just like with general purpose office tools. The resulting meta model is merely a means to an end. Primarily, BitKit semantically combines equally looking elements by deriving a common associated entity. After a meta model is inferred and, for instance, the color of such an element is changed the color of every other (equally looking) element is adapted accordingly. Due to the office tool intention of BitKit, the generated meta model is not intended to be processed in any further way. Consequently, its quality is not considered as well.

Another approach is proposed in [25]. Like BitKit, it is also restricted to graphical DSLs. Nevertheless, we adopt their general idea for applying patterns when inferring a meta model. That meta model (which represents the abstract syntax as stated by the author) highly corresponds to the concrete syntax as well. This correspondence is obvious when investigating another publication of Cho and Gray. In [26], they introduce some design patterns well suited for meta models. However, the presented patterns are very specific for graphical DSLs and hence not universally valid. That can be verified when comparing these patterns to the meta models for visual languages defined in [27]. In contrast to our approach, they directly apply design patterns wherever possible. Owing to the visual information, they can resort to additional domain

knowledge, which we do not have on hand. However, our recommendation framework can also be applied to their meta models and hint to artefacts of these meta models with potential for further refinement.

In parallel to our research, a similar approach has been published in [14]. They infer a meta model from example models, which are specified using a predefined textual concrete syntax. From their approach, we adopted the idea of providing recommendations such that a meta model's quality can be increased. Since [14] is rather an overview paper, the authors do not provide detailed solutions how detection of recommendations works. In this article, we minimized that gap and presented some concrete methods how constellations of meta model elements with potential for refinement can be identified.

## VIII. OUTLOOK

The presented rapid design approach works well for meta models, which are formulated using a linguistic meta language as concrete syntax. For entire DSLs, further effort is necessary since each DSL features its own concrete syntax whose specification process should also follow the proposed rapid design principle. For sketching textual concrete syntaxes, we already published a method in [28].

Our next step is to combine the meta model derivation approach presented in the current paper with the construction of custom concrete syntaxes. Beyond textual syntaxes, we also contemplate to support graphical DSLs.

To conclude, the overall goal is developing a system, which fosters the rapid design and usage of all artefacts DSLs consist of. This means that the intended system provides a seamless integration of free and stringent modelling when working with meta models and even entire DSLs.

## ACKNOWLEDGMENT

This article was authored in the context of the project "Kompetenzzentrum für praktisches Prozess- und Qualitätsmanagement" (KpPQ) funded by "Europäischer Fonds für regionale Entwicklung" (EFRE). So, we thank this institution, which has kindly facilitated our work.

## REFERENCES

[1] B. Roth, M. Jahn, and S. Jablonski, "A method for directly deriving a concise meta model from example models," in *Proceedings of the 5th International Conferences on Pervasive Patterns and Applications*, 2013, vol. 5, no. 1, pp. 52–58.

[2] T. Clark, P. Sammut, and J. Willans, *Applied Metamodelling: A Foundation for Language Driven Development*. CETEVA, 2008, p. 227.

[3] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling Full Code Generation*, 1st ed. John Wiley & Sons, 2008, p. 444.

[4] E. M. Gold, "Language identification in the limit," *Inf. Control*, vol. 10, no. 5, pp. 447–474, 1967.

[5] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, "MARS: a metamodel recovery system using grammar inference," *Inf. Softw. Technol.*, vol. 50, no. 9–10, pp. 948–968, 2008.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[7] B. Roth and M. Jahn, "Model Workbench," 2013. [Online]. Available: http://www.ai4.uni-bayreuth.de/de/research/projects/003_ModelWorkbench/index.html. [Accessed: 28-Jan-2013].

[8] J. Odell, *Advanced object-oriented analysis and design using UML*. Cambridge University Press, 1998.

[9] C. Atkinson and T. Kühne, "Concepts for comparing modeling tool architectures," in *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, 2005, pp. 398–413.

[10] B. Volz and S. Jablonski, "Towards an open meta modeling environment," in *Proceedings of the 10th Workshop on Domain-Specific Modeling*, 2010.

[11] C. Atkinson and T. Kühne, "Meta-level independent modelling," in *Proceedings of the International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*, 2000, pp. 12–16.

[12] B. Volz, "Werkzeugunterstützung für methodenneutrale Metamodellierung," University of Bayreuth, PhD thesis, 2011.

[13] M. F. Bertoa and A. Vallecillo, "Quality attributes for software metamodels," in *Proceedings of the 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2010.

[14] J. Sánchez-Cuadrado, J. De Lara, and E. Guerra, "Bottom-up meta-modelling: an interactive approach," in *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, 2012, pp. 3–19.

[15] D. Flanagan, *JavaScript: The Definitive Guide*, 6th ed. Sebastopol, CA: O'Reilly Media, Inc., 2011, p. 1078.

[16] B. Liskov, "Data abstraction and hierarchy," *ACM SIGPLAN Not.*, vol. 23, no. 5, pp. 17–34, 1988.

[17] G. Singh, "Single versus multiple inheritance in object oriented programming," *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, 1994.

[18] M. Herrmannsdoerfer, S. D. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *Proceedings of the 3rd International Conference on Software Lanugage Engineering*, 2010, pp. 163–182.

[19] M. Jahn, B. Roth, and S. Jablonski, "Remodeling to powertype pattern," in *Proceedings of PATTERNS 2013*, 2013, pp. 59–65.

[20] J. Bloch, *Effective Java*, 2nd ed. Upper Saddle River, New Jersey: Addison-Wesley Longman, 2008, p. 384.

[21] M. Mernik, D. Hrncic, B. R. Bryant, A. P. Sprague, J. Gray, Q. Liu, and F. Javed, "Grammar inference algorithms and applications in software engineering," in *22th International Symposium on Information, Communication and Automation Technologies*, 2009, pp. 1–7.

[22] F. King-Sun and T. L. Booth, "Grammatical inference: introduction and survey - part I," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 3, pp. 343–359, Mar. 1986.

[23] Q. Liu, B. R. Bryant, and M. Mernik, "Metamodel recovery from multi-tiered domains using extended MARS," in *Proceedings of the 34th IEEE Annual Computer Software and Applications Conference*, 2010, pp. 279–288.

[24] M. Desmond, H. Ossher, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, and S. Krasikov, "Towards smart office tools," in *SPLASH 2010 Workshop on Flexible Modeling Tools*, 2010.

[25] H. Cho, J. Gray, and E. Syriani, "Creating visual domain-specific modeling languages from end-user demonstration," in *ICSE Workshop on Modeling in Software Engineering*, 2012, pp. 22–28.

[26] H. Cho and J. Gray, "Design patterns for metamodels," in *Proceedings of the SPLASH '11 Workshops*, 2011, pp. 25–32.

[27] P. Bottoni and A. Grau, "A suite of metamodels as a basis for a classification of visual languages," in *Symposium on Visual Languages and Human Centric Computing*, 2004, pp. 83–90.

[28] B. Roth, M. Jahn, and S. Jablonski, "On the way of bottom-up designing textual domain-specific modelling languages," in *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling*, 2013, pp. 51–55.