A Method for Directly Deriving a Concise Meta Model from Example Models

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

***Abstract*—Creating 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. They take graphical models as input and primarily analyze graphical properties of the utilized shapes to derive an appropriate meta model. Instead of that, 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 conciseness, we aim to apply so-called language patterns like inheritance and enumerations. For it, the applicability of those patterns is analyzed concerning the available information gathered from the underlying model examples. Furthermore, algorithms are introduced which apply the different patterns to a given meta model.**

Keywords-meta model derivation; meta model inference; conciseness of meta models; pattern recognition; language patterns; inheritance

I. INTRODUCTION

Manually creating domain-specific languages (DSL), especially with a concise meta model as abstract syntax, is a complex task [1]. Besides an abstract syntax, a typical DSL also consists of a concrete syntax and a set of semantic rules (constraints) [1]. In this paper, the focus lies on the abstract syntax defined by a meta model. For defining such a meta model, new development methods have emerged. Those methods focus on deriving or inferring a meta model from a given set of example models [2, 3]. However, they only marginally consider the conciseness aspect of the resulting meta model (if at all). According to [4], this is a very important quality criteria of meta models. Therefore, our primary goal is to obtain a meta model with a high degree of conciseness. To achieve this, a typical solution is to apply language patterns like single inheritance, multiple inheritance and enumerations to a constellation of meta model entities (for more information about conciseness see section III).

Since the resulting meta model should represent the abstract syntax of a DSL another important goal of our approach is to derive a meta model which highly corresponds to concepts describing the domain. Hence, we have to gain the domain entities' instances from the model examples. Such instances directly can be modeled when using the Open Meta Modeling Environment (OMME) introduced by Volz et al. in [5]. Consequently, the paper at hand originates in the context of OMME. In the following sub section, we shortly explain the relevant characteristics of this platform.

A. The Open Meta Modeling Environment

OMME is an Eclipse-based meta modeling tool [6] that allows developers to define their own modeling language. It goes far beyond the capabilities of competing tools with respect to its support for advanced language patterns (e.g. Powertypes [7]). Its implementation is based on the Orthogonal Classification [8] and uses Clabjects [9] for representing concepts of a model (the term "concept" in the context of OMME always means a Clabject). Hence, OMME provides a Linguistic Meta Model (LMM) and interprets (meta) models at runtime in order to emulate a concrete textual syntax.

Below, we predominantly limit ourselves to concepts which can act as both, types and instances. As a type (also called a meta concept), a concept declares attributes whereas as an instance (also called an instance concept), a concept contains assignments each of which may be associated with an attribute. If such an association exists the target attribute must be declared by the type (meta concept) of the assignment's owner. Attributes and assignments can be divided into literal and referential ones depending on their respective type. OMME supports the following literal types: boolean, integer, double and string. 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. While modeling using the LMM, the applicable language patterns can be selected according to a user's needs (e.g., enabling or disabling multiple inheritance). Below, each suchlike configuration is called a modeling context.

B. Fundamental assumption on equally named elements

The most important assumption we take is that equally named elements (types of 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 the conciseness of a given meta model can be enhanced. Both

 ISBN: 978-1-61208-276-9

approaches presented in section V infer graphical DSLs and follow a comparable principal. They state that two shapes correspond to each other if their graphical properties are identical.

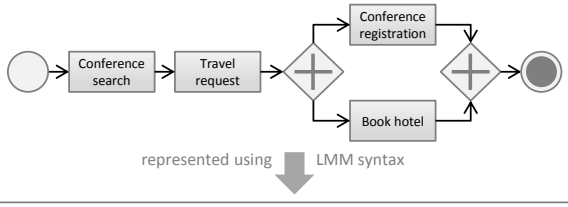
II. EXAMPLE MODEL

As an example model we constructed the process shown in Fig. 1 using two different concrete syntaxes. The top part shows the graphical representation with nodes and directed edges. It is just depicted for a better comprehensibility since a graphical process model is easier to understand than a textual one. In the right, the same model is written down using the concrete textual syntax given by the LMM.

Below, we focus on the textual representation because it directly uses constructs of the LMM. Since the LMM syntax is quite similar to the one of popular object-oriented programming languages it is easy to read for software developers and modelers. The mapping rules between both representations lie beyond the scope of this paper, so the following mapping is taken for granted: The circle node on the left is considered as `Start` concept with identifier `S`. It contains an assignment `next` which refers to concept `P1` and represents a successor relationship. `P1` to `P4` are specified as Process concepts. Each of them has a title and also a `next` assignment. `A1` and `A2` represent instances of concept `And`. Both again contain a next assignment. However, assignment `next` of `A1` holds two references to `P3` and `P4`. The last circle on the bottom represents the process models `Exit`. It contains no further assignments. The arrows between the different nodes can be seen as successor relationships which are always mapped to according `next` assignments.

III. CONCISE META MODEL USING LANGUAGE PATTERNS

One important goal in meta modeling is keeping meta models concise [4]. Therefore, models need to be as small as possible, i.e., they should completely describe their according domain with as few constructs as possible. Achieving this is a general problem when building meta models. For instance, the authors of the newly published version 2.5 of UML have focused on simplifying the corresponding meta model [10].




```
Start S {
     next = P1
}

Process P1 {
     title = "Conference Search"
     next = P2
}

Process P2 {
     title = "Travel request"
     next = A1
}

And A1 {
     next = P3, P4
}

Process P3 {
     title = "Conference registration"
     next = A2
}

Process P4 {
     title = "Book hotel"
     next = A2
}

And A2 {
     next = E
}

Exit E {
}
```

Figure 1. Example model visualized using two different concrete syntaxes

Making a meta model concise can be accomplished by applying so called language patterns to suitable constellations of meta elements [11]. In literature, it is not exactly specified how a suitable constellation looks like. There are only suggestions in form of best practices or guidelines when to apply a certain pattern (comparable to the applicability of design patterns [12]). Because these guidelines are suggestions they are mostly formulated quite imprecise with a subjective touch. Most guidelines base on domain-specific background knowledge (e.g., the "is a"-statement mentioned in the following sub section A for using single inheritance). In general, such information is not available. Hence, we have to rely on the information provided by the model examples as well as the structure of the derived meta model (i.e., attributes and their referential or literal types).

In the following, three typical language patterns that are supported by OMME and partially many different other modeling frameworks (e.g., EMF, MetaGME, eMOFLON) are presented. For enumerations, we do not elaborate further because their usage is straightforward. They basically allow for restricting the value range of an attribute to a few predefined literals.

A. Single inheritance

Single inheritance is a well-known and widespread language pattern stemming from the field of object-oriented programming languages. There, it allows for introducing generalization/specialization hierarchies on classes. The key feature necessary for our approach is that a specialized class inherits all fields of its super class.

The most common rule for introducing a specialization relationship is: if an "is a"-relationship can be identified between two classes [13] (or entities like stated in [14]) the source of this relationship specializes the target. To identify this kind of relationship, background knowledge about the domain is required which cannot be directly expressed through the model example(s). Therefore, we follow the proposal of [15] and interpret a set of shared attributes as indicator for an inheritance relation. In some cases, for a given model example the introduction of a specialization relationship is indispensable. This occurs if an attribute is intended to reference two or more different classes. Then, those referable classes need a common super class which has to be the type of the aforesaid attribute. An example for that is demonstrated in section IV.B step 3. This additional information can only be retrieved from the model examples and not directly from the meta model. That is the case because merely in instances different concepts may be assigned to attributes (according to their respective types). A referential attribute, however, always expects exactly one type.

Another important topic when using inheritance is a rather flat generalization hierarchy. Otherwise, the meta model gets quite complex and thus its comprehensibility suffers.

B. Multiple inheritance

Multiple inheritance is often criticized as risky because of potentially occurring problems as stated by Singh in [16] (e.g., name collision and repeated inheritance). Hence, we only utilize multiple inheritance to meet addressability constraints

 ISBN: 978-1-61208-276-9

found within the original model example(s). Addressability means the two possible referencing aspects, namely "a concept is referenced by another one" and "a concept refers to another concept". An adequate example can be found in section IV.C.2) where an algorithm is proposed for applying the multiple inheritance language pattern to a given meta model. This restriction protects also from over-generalizing the resulting meta model.

IV. META MODEL DERIVATION

When deriving a meta model from a model example, the directly recognizable constraints need to be softened in some way. Otherwise, solely the provided model example can be re-modeled without any differences. This softening behavior needs to be highly configurable since the statement whether a meta model is concise or not is always subjective. Therefore, our prototypical implementation provides many according parameters which allow for fully customizing the derivation behavior. However, the given default settings represent the notion of a concise meta model based on our experiences and best practices.

In the following, we introduce our direct method for deriving a concise meta model out of one (or more if available) model example(s). Direct method means that we directly work with constructs given by the LMM. In the first instance we refer to concepts, assignments and attributes. The whole method can be divided into two main parts, according to the necessity whether applying language patterns is required or not:

- Bottom-up part: for each found unique type a separate meta concept is created with all required attributes. After that, language patterns are applied that are mandatory for obtaining a valid meta model as defined by the LMM's semantics.
- Conciseness part: analysis of the generated meta model to find constellations of concepts to which further language patterns can be applied. These constellations are identified according to the statements about the particular patterns in section III.

A. Reusable sub algorithms

Below, three sub algorithms are presented that are reused at different places. So, their functionality is described once and referenced wherever needed.

1) Merging a set of types using generalization

The sub algorithm "merging types using generalization" has the task that for a given set of types, one common super type has to be determined (without moving contained attributes from the input types to a new common super type). Its functionality correlates to the one provided by the model evolution operations "extract super class" and "fold super class" described in [17]. Nevertheless, both operations always base on at least one common feature (in our terms: one common attribute) which is not the case for our algorithm.

The algorithm works as follows: Receiving a set of input types `ITs`, for each type `IT` the routine collects its super types and add them to the set `STs`. Those super types `STs` are analyzed whether each one of their specializations `SPs` (sub types) is contained by the set of input types `ITs`. If so the particular super type `ST` is a merging candidate `C`. After processing the input types, all found candidates `Cs` are merged to one common super type `CST` (disjunction). In case no super type candidates `Cs` are found, a new common abstract super type `CST` is generated. Finally, over `ITs` is iterated again. Thereby all specialization relationships from the type to any candidate `C` are removed. In place of that, a new specialization relationship is inserted from the type `IT` to the new common super type `CST`. As a cleanup, each super type `ST` that is no longer specialized is removed from the meta model. Furthermore, all references to the former super types `STs` (if exists) are replaced by according references to the new common one (`CST`). In addition to this informal description of the algorithm's functionality, Figure 2 gives an overview by means of a corresponding flow diagram.

After performing this algorithm, the resulting common super type may contain attribute duplicates. They may appear when merging several super type candidates to one common super type. Due to reusability reasons, it is not in the scope of this algorithm to resolve this inconsistency. That has to be done afterwards.

2) Elimination of attribute duplicates

Another frequently reused sub routine is "eliminating attribute duplicates". This algorithm takes a concept with inconsistent content as input. Inconsistency is enunciated by several equally named attributes which need to be merged to one single attribute.

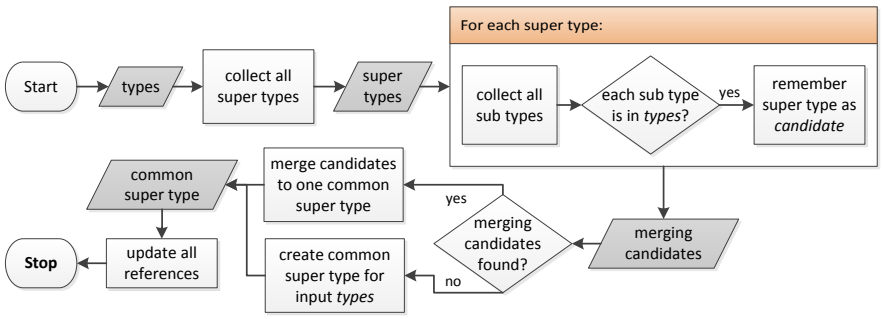


Figure 2. Flow diagram for "merging types using generalization" algorithm

Handling different cardinalities is quite easy. They are always widened to flexibility and consequently less limitations (e.g., 1 and 1..* turn to 1..*, 0..1 and 1..* to 0..*).

However, before addressing the type merging part, attributes have to be split up according to their kind, namely referential or literal. It is important to notice again that enumerations are regarded as literal attributes, too. The fork is necessary because referential attributes may lead to an indispensable introduction of a generalization hierarchy.

For instance, imagine a source concept with two equally named attributes whose types are referring to two different target concepts. In order to maintain the possibility of referencing instances of both target concepts within an instance of the source concept, the target concepts need a common super type. Thereby, for a set of equally named attributes the attribute types are extracted. If these types refer to different meta concepts for all those concepts a common abstract super concept is created and specialized by them (using sub algorithm 1)). Afterwards, only one of the original attributes is kept and its type (the referenced meta concept) is changed to the new common super concept. For another concrete example, see Figure 4 and Figure 5.

Eliminating or at least handling several equally named literal attributes happens in a different way. For it, we conceive three alternative strategies which may be configured as mentioned at the beginning of this section. The first one just informs the user about these ambiguities. The second strategy renames all duplicate attributes by means of a predefined rule (e.g., appending ascending numbers to the attributes' names). That also leads to small modifications within the model example(s) because the respective assignments must be updated as well. Using the third and last alternative conforms to our overall intention to a greater extent. We stated above that equally named constructs are considered to correspond to the same artifacts at domain side. Therefore, the third strategy merges the duplicate literal attributes based on type widening. This concept is comparable to the one of popular programming languages like Java and C#. Hereby, we allow type widening for all literal data types (enumerations included). When applying it to two different types then always the one with a greater value range is chosen. The ascendant order of the literal types according to their value range is as follows: boolean, enumeration, integer, double, string.

It may also occur that there are equally named referential and literal attributes at the same time. In this case it is obvious that only the first two strategies are expedient (i.e., inform the user or rename the concerning attributes and assignments). Due to the different inherent intents of literal and referential attributes, merging is not a valid option.

3) Elimination of multiple inheritance

Executing the task "eliminating multiple inheritance" is required if some concepts specialize more than one super type but multiple inheritance is not available in the current modeling context. At first, the according algorithm looks for concepts `Ts` which specialize at least two other concepts (set of all super types `STs`). Next, it iterates over all found concepts `STs`. For each concept `ST`, it selects all concerning sub concepts `SPs` that extend one or more super types specialized by `T`. Moreover, algorithm 1) is called by delivering all specializations `SPs` (`T` incl.) as input data.

Merging types this way may lead to attribute duplicates. They have to be eliminated by algorithm 2). Since its execution could again produce more than one super type per concept cyclic invocation of both algorithms may be necessary. This cycle will definitely terminate. At the latest this occurs when one global super type is found which is used as generalization for all other concepts.

For eliminating multiple inheritance, extending the inheritance hierarchy about a further level is another conceivable solution. However, the solution is not universally valid (like the chosen solution stated above) because it cannot be applied to each constellation of concepts. For instance, that is the case if there are many different attributes which are mutually used within various concepts.

B. Bottom-up algorithm

The initial bottom-up algorithm (Figure 3) is considered as obligatory for deriving an initial meta model. For this algorithm, the (instance) concepts of one or more example models are taken as input data. The algorithm itself can be divided into four main steps.

Within the first step, for each uniquely identified type in the model example(s) a separate meta concept is created. Applied to the example from section II the unique meta concepts `Start`, `Process`, `And` and `Exit` are derived.

The second step infers attributes according to the assignments specified in the particular instantiating concepts. Hereby, for each assignment a corresponding attribute is created. This attribute takes over the name, the type and the cardinality of the assignment. In doing so, the cardinality's lower bound is set to 1 if each instance of the same type contains such an assignment, otherwise to 0. The upper one is set to 1 if every time only one value is assigned, else * is chosen. For literal assignments, the type can be directly read off because this recognition task is carried out by the LMM's parser. Handling referential assignments is more complex. If solely one concept is referenced then its type is directly borrowed from it. Otherwise, for each referenced concept its

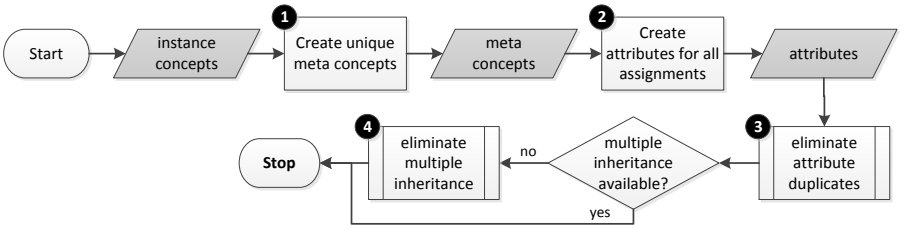


Figure 3. Coarse-grained flow diagram for the bottom-up algorithm

 ISBN: 978-1-61208-276-9

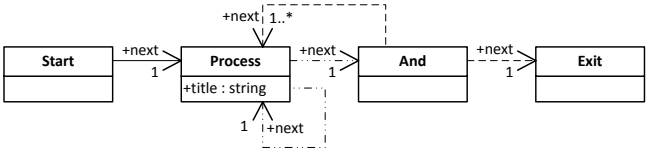


Figure 4. Meta model for the above example after executing step 2

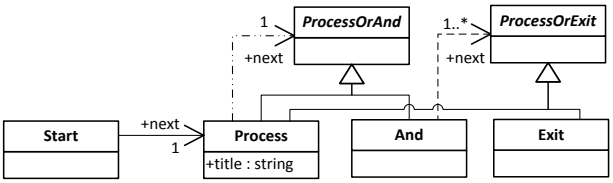


Figure 6. Meta model for the above example after executing step 3

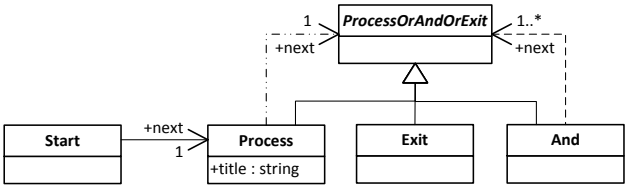


Figure 7. Meta model for the above example after executing step 4

type is detected individually. In case different outcomes occur, for every type a separate attribute is generated.

After finishing step 2, the meta model for the aforementioned example looks like the one depicted in Figure 4. Therein, the two `next` attributes of `Process` as well as the two of `And` must be merged in some way. This is done by invoking sub algorithm IV.A.2).

Thereby, two abstract super concepts are generated, namely `ProcessOrExit` and `ProcessOrAnd`. The so modified meta model is shown in Figure 5. The style of the arrows symbolizing the referential `next` attributes is the same as the style of the arrows which represent their sources in Figure 4.

Step 4 is merely required if multiple inheritance is disabled for the current modeling context. Then `Process` may only specialize one super concept. To achieve this, sub routine IV.A.3) is invoked. When applying it to the meta model from Figure 5, `ProcessOrAnd` and `ProcessOrExit` are merged to `ProcessOrAndOrExit` as depicted in Figure 6. Beyond that, the specialization relationships of `Process`, `And` and `Exit` must now point to `ProcessOrAndOrExit`. The same is true for the referential next attributes which refer to `ProcessOrAnd` respectively `ProcessOrExit`.

C. Technical applicability of language patterns

Below, for each supported language pattern a separate conciseness algorithm is presented that applies this pattern to a given meta model. Every conciseness algorithm requires so called "corresponding attributes" as input data. Thereby, two different correspondences need to be distinguished. As stated in the introduction, equally named attributes are intended to have the same meaning according to the particular domain. In other words, different attributes which correspond to each other always carry an identical name. The second correspondence bases upon the first one because sets of such corresponding attributes may be again subsumed to a superior set. In contrast, this correspondence does not base on the attributes' names but on their owners. Hence, two sets of corresponding attributes correspond only if each attribute of one set has a counterpart in the other set which both exhibit the same owner.

Before applying any language pattern, these corresponding attributes have to be determined and the according data structure must be built up. For it, all equally named attributes are put into appropriate sets. Depending on the underlying configuration, the attributes' types and cardinalities are regarded or ignored. Afterwards, the superior sets are created by extracting subsets from the former ones whose attributes meet the aforementioned owner criterion. This calculation task can be simplified by sorting the attributes within the former sets by their owners.

1) Single inheritance

The conciseness algorithm that applies single inheritance (Figure 7) can be split up into two variants. The first variant (yes-path) takes one of the input attributes' owner as common super type, whereas using the second variant (no-path) a new common super type is built up.

Choosing the particular variant bases on information gathered in step 1. Herein, the incoming attributes' owners are scanned for a concept which can be taken as common super type. Such a concept must declare all common attributes which can then be inherited by any sub concept (step 3).

In step 4, all corresponding attributes from the sub concepts are moved to the common super concept. This results in an inconsistent meta model because several equally named attributes occur within the super type. Then, step 5 invokes sub routine A.2) which resolves this inconsistency. However, execution of step 5 may bring multiple inheritance to the meta model (see section IV.B and especially Figure 5 for an according example). This potential problem is addressed by step 6 that encapsulates sub routine A.3).

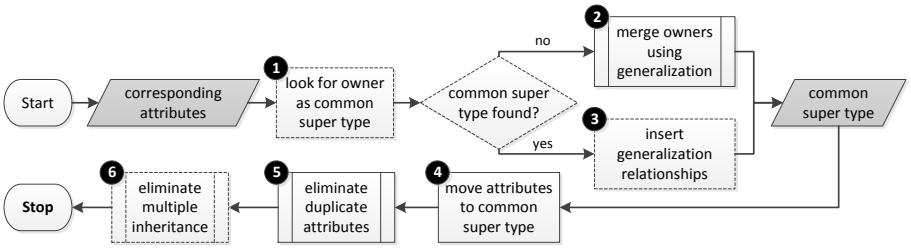


Figure 5. Flow diagram for the conciseness algorithm that applies single inheritance

Following the second variant (no-path) will be done if in step 1 no common super type is found and hence, one has to be determined. Then, step 2 calls the aforementioned sub routine IV.A.1). Applying it (the subsequent steps included) to the final meta model generated by the bottom-up algorithm (section IV.B), all three `next` attributes are delivered as input to the algorithm described above. Since none of the attributes' owners can be used as common super type those owners have to be merged accordingly. This has led to a new super type called `StartOrProcessOrAnd`. Afterwards, the `next` attributes are moved to this new super concept and merged (their common type is set to `ProcessOrAndOrExit`). Now, `Process` as well as `And` specialize two concepts, namely `StartOrProcessOrAnd` and `ProcessOrAndOrExit`. Due to the requirement of using single inheritance, both super concepts are merged to a single concept named `ProcessOrAndOrExitOrStart` and all references to the former ones are updated.

Apparently, `Exit` may also have a successor now, which was not intended by the model example. As explicated in section IV.A.3) (third sub algorithm), that is a negative side effect when restricting to single inheritance. This problem typically is solved by integrating a constraints system. When using a suchlike system, however, the brought constraint language needs to be studied first. All in all, that decreases the comprehensibility of the generated meta model and thus has a negative impact on its conciseness.

2) Multiple inheritance

Due to the aforementioned restriction to addressability constraints, the algorithm for applying multiple inheritance only has to consider referential attributes. Consulting the example from section II, concepts of type `Start` may never be "referenced by" any other concept. In doing so, an instance of `Exit` may not be able to have a successor by "referring to" any target concept (via `next`). However, reducing the number of equally named attributes is still our base intent. Keeping those two objectives in mind and applying them to the meta model depicted in Figure 5, the resulting meta model will look like the one visualized by Figure 8. Here, the two different concerns mentioned above ("references by" and "refer to") are implemented by means of a separate generalized concept. The first one is represented by `ProcessOrAndOrExit`, while the "refer to" aspect is established via `StartOrProcessOrAnd`.

Consequently, an appropriate algorithm needs to regard both aspects. However, utilizing the knowledge about the algorithm for applying single inheritance, the solution for multiple inheritance is similar. We directly take the algorithm for single inheritance and remove some superfluous steps.

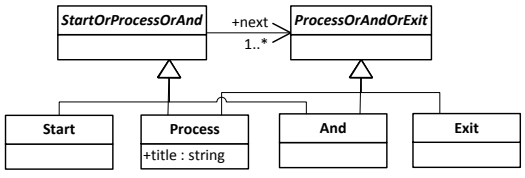


Figure 8. Meta model after applying the multiple inheritance language pattern

These superfluous steps are marked in Figure 7 by a dashed border. So, the resulting algorithm merely contains steps 2, 4 and 5. Besides, it only accepts referential attributes as input.

3) Enumerations

The conciseness algorithm for inferring enumerations is simpler than the two for applying single or multiple inheritance. Nevertheless, it requires more information as input, namely all assignments belonging to an attribute or a set of corresponding attributes. The selection whether to choose a single attribute or a set of corresponding attributes must be taken by the user in a previous configuration step. However, this has no impact on the main flow of the algorithm. Using a set of attributes just means to process more according assignments than with only one single attribute. From these assignments, the values are used to determine the resulting enumeration's literals. Hence, only literal attributes of type string are supported as input.

Whether an enumeration is generated or not depends on the diversity of values held by the different assignments. If there are merely a few values which are repeatedly assigned to that attribute(s) a new enumeration is derived. The varied values are taken as unique literals for this enumeration. Accordingly, the assignments have to be updated with the new literal values as well. So, when applying the enumeration language pattern the underlying model examples suffer small modifications. That is why this algorithm has to be executed before running the two others (for single or multiple inheritance).

V. 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 [18]. Thereby, many textual sentences (ideally positive and negative samples) are analyzed to infer a grammar [19].

In current research, the Metamodel Recovery System (MARS) is one prominent representative for meta model reconstruction [20]. 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 only two research groups that generate a meta model by deriving it from very few model examples. BitKit as one representative has a rather different intention [21]. Its authors aim at supporting the pre-requirements analysis of software products by allowing to

 ISBN: 978-1-61208-276-9

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 [22]. 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 correspondency is obvious when investigating another publication of Cho and Gray. In [23] 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 [24]. In contrast to our approach, they mix the two identified main parts (section IV) when inferring a meta model. Hence, applying design patterns is strongly enmeshed in the bottom-up part. Thus, using our conciseness algorithms instead of their proposed "design pattern"-based approach is not possible without great effort.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented a method for directly deriving a concise meta model from a small set of example models. To increase the conciseness of the resulting meta model, language patterns are applied to an appropriate constellation of meta concepts. Due to page limitations, we focused on widespread language patterns like inheritance and enumerations. As mentioned in section I.A, there are further language patterns (e.g. Powertypes) supported by OMME. Thus, we currently develop or extend the above conciseness algorithms for those patterns. Afterwards, we explore design patterns that can be applied similar to the way described above (but not only for visual languages like in existing solutions).

Our approach of automatically applying language patterns to meta concepts can also be reused for refactoring activities in modern IDEs like Eclipse or Visual Studio. Hereby, classes are considered as concepts whereas their fields are regarded as attributes. Taking the same assumptions as described in section I.B and providing appropriate configuration options, the presented conciseness algorithms can be taken for applying particular language patterns to a collection of classes. In future research, we also will deal with this topic in more detail.

REFERENCES

[1] T. Clark, P. Sammut, and J. Willans, "Applied metamodelling: a foundation for language driven development," CETEVA, 2008.

[2] H. Ossher, R. Bellamy, I. Simmonds, D. Amid, A. Anaby-Tavor, M. Callery, M. Desmond, J. de Vries, A. Fisher, and S. Krasikov, "Flexible modeling tools for pre-requirements analysis: conceptual architecture and research challenges," in *Proceedings of OOPSLA 2010*, vol. 45, 2010, pp. 848–864.

[3] H. Cho, "A demonstration-based approach for designing domain-specific modeling languages," in Proceedings of SPLASH 2011, 2011, pp. 51–54.

[4] M. F. Bertoa and A. Vallecillo, "Quality attributes for software metamodels," Proceedings of QAOOSE, 2010.

[5] B. Volz and S. Jablonski, "Towards an open meta modeling environment," Proceedings of the 10th Workshop on Domain-Specific Modeling, 2010, pp. 17-1–17-6.

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

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

[8] C. Atkinson and T. Kühne, "Concepts for comparing modeling tool architectures," Model Driven Engineering Languages and Systems, 2005, pp. 398-413.

[9] C. Atkinson and T. Kühne, "Meta-level independent modelling," International Workshop Model Engineering in Conjunction with ECOOP 2000, 2000, pp. 12-16.

[10] S. Covert, "OMG's Unified Modeling Language (UML) Celebrates 15th Anniversary," 2012. [Online]. Available: http://www.omg.org/news/releases/pr2012/08-01-12-a.htm.

[11] C. Atkinson and T. Kühne, "The role of metamodeling in MDA," Proceedings of the International Workshop in Software Model Engineering 2002, 2002, pp. 67-70.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.

[13] Microsoft Corporation, "When to Use Inheritance," 2008. [Online]. Available: http://msdn.microsoft.com/en-us/library/27db6csx(v=vs.90).aspx.

[14] M. Gogolla and U. Hohenstein, "Towards a semantic view of an extended entity-relationship model," ACM Transactions on Database Systems, 1991, pp. 369-416.

[15] R. Elmasri and S. B. Navathe, Fundamentals of database systems, 3. A. Amsterdam: Addison-Wesley Longman, 2000.

[16] G. Singh, "Single versus multiple inheritance in object oriented programming," ACM SIGPLAN OOPS Messenger, 1994, pp. 30-39.

[17] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," Software Language Engineering, pp. 163–182, 2011.

[18] 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 Proceedings of the 9th International Colloquium on Grammatical Inference, 2009, pp. 1–7.

[19] F. King-Sun and T. L. Booth, "Grammatical Inference: Introduction and Survey - Part I," IEEE transactions on pattern analysis and machine intelligence, vol. 8, 1986, pp. 95-111.

[20] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, "MARS: A metamodel recovery system using grammar inference," Information and Software Technology, vol. 50, 2008, pp. 948–968.

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

[22] H. Cho, J. Gray, and E. Syriani, "Creating visual Domain-Specific Modeling Languages from end-user demonstration," Modeling in Software Engineering, 2012, p. 22-28.

[23] H. Cho and J. Gray, "Design patterns for metamodels," Proceedings of SPLASH 2011, 2011, pp. 25-32.

[24] 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.

 ISBN: 978-1-61208-276-9