Instance Modeling Assisted by an Optional Meta Level

Riccardo Solmi
Whole Factory, Italy
solmi.riccardo@gmail.com

Abstract

We believe that programming is mainly a linguistic process concerning the development of the language abstractions better suited to deal with a given problem domain. The main responsibility of a linguistic system is to capture and incorporate the knowledge of domain experts, while trying to minimize the meta level efforts, thus allowing users to concentrate on modeling activities.

While a meta level is necessary in order to write instances, it is possible to define a generic meta level capable of representing any specific entity.

We introduce an instance modeling language, Entities, combining a rich graphical notation, an optionally typed structure, and composability with other domain specific languages. The visual expressivity is comparable to a mindmapping tool, and makes it best suited for knowledge representation domains.

The optional typing enables an exploratory, bottom up approach to metamodeling. The composability with strictly typed languages makes modeling a much more flexible experience.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications – Elicitation methods (e.g., rapid prototyping), Languages, Tools; D.2.2 [*Software Engineering*]: Design Tools and Techniques – Evolutionary prototyping; D.2.6 [*Software Engineering*]: Programming Environments – Graphical environments, Interactive environments; D.2.13 [*Software Engineering*]: Reusable Software – Domain engineering; D.3.2 [*Programming Languages*]: Language Classifications – Specialized application languages

General Terms Design, Human Factors, Languages

Keywords Bottom up metamodeling, End user programming, Domain-specific modeling, Domain-specific languages, Language Workbenches, Whole Platform

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

DSM'16, October 30, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4894-2/16/10...$15.00
http://dx.doi.org/10.1145/3023147.3023156

1. Introduction

Domain specificity can be reached in every aspect of the definition of a domain specific language: structure, notation, persistence, tooling, and other semantics. Having a specific definition for an aspect may be regarded either as an advantage or as a limitation depending on the usage scenarios considered. For instance, a generic implementation of a persistence may speed up the modeling process and may even avoid the need of defining a specific persistence if it is not a user requirement. While a specific notation is regarded as a more attractive choice for a domain expert, the cross language uniformity granted by a generic one may turn out to be more effective for some activities of a language designer.

A few domain specific modeling tools enable multiple definitions for some aspects of a language and even provide generic implementations for them.

The metamodel is, in general, a notable exception because, in a model driven approach, it has a central role even at the implementation level. A metamodel tailored to the specific abstractions and structure of a given domain has many benefits including granted conformity of an instance to its metamodel. Unfortunately, there are scenarios in which it is better to trade conformity off for flexibility and allow invalid model instances.

In order to implement our solution in the form of a domain specific language called `Entities`, we used the graphical and projective language workbench tool that is part of the open source project **Whole Platform** [1] Additional information on the Whole Platform and a comparative evaluation with other alternative tools can be found in [2].

In the sections that follow, we present an overview of the available approaches to domain specific modeling in order to motivate the need of a different solution, then we present the new language `Entities` to perform flexible instance modeling together with an example, and finally we outline other usage scenarios.

2. Approaches that Need Meta Modeling

There are multiple approaches to domain specific modeling that sooner or later require explicit meta modeling tasks. This assumes that there is a reasonable meta model for each

domain, and that there is always someone with language design skills.

2.1 Meta Modeling First

There are several useful approaches to define a language starting by modeling its meta level.

The Whole Platform, for instance, provides multiple domain languages to address the different aspects of a language definition. Luckily, an explicit meta modeling activity can be limited to just one aspect of the language definition, the others can be derived, postponed, or omitted by relying on generic implementations.

The language designer simply defines either the metamodel, or a grammar, or a translational semantics, in order to get an instanceable language for the domain expert.

2.2 Continuous Cycling through Meta Levels

By adding interpreting or dynamic compiling support to the meta modeling languages you may get a form of live programming called live language development in [3]. A live programming approach allows a language designer to edit the meta level definitions of a language while playing with its instances. The improvement in speed is remarkable, but, to take full advantage of it, it is necessary to provide a migration facility for the model instances.

This approach can allow a language designer and a domain expert to work side by side, in a sort of pair programming, where the former try to keep pace with the needs of the latter.

2.3 Instance Modeling First

There are multiple approaches to enable instance modeling before performing any explicit meta modeling activity.

For instance, it has been proposed to introduce a graphical drawing language [5] [6] [10] [11] or a text editing language [8], or an instance model manipulation language [4] [7]. Either way, by using additional constructs to mark parts of the instance model with meta hints or by tracing user interactions, it is possible to derive at least a partial metamodel suitable for writing and validating the given model instances.

Note that the domain expert, at the beginning of the process, uses a tool to write some examples that are used by the language designer to produce the real language. Only later, and usually with a different tool, the domain expert can write the desired instances.

In [9] the examples are made an integral part of the definition of a language on par with the metamodel and they are the driving force of the entire language development process.

3. Standalone Instance Modeling

We think that there are many domains for which it makes little sense to explicitly define a specific metamodel. For instance, the presence of a generic metamodel inside of the mind mapping tools is not perceived as a limitation and it has not prevented the spread of mind mapping templates, that represent a sort of domain language.

firstName	Riccardo			
lastName	Solmi			
company	Whole Factory srl			
address	streetAddress	Via 123		
	city	Bologna		
	state			
	country	Italy		
	postalCode	40033		
phoneNumber	type	home	type	fax
	number	012 3456789	number	987 6543210

Figure 1. Nested tables (JSON specific notation)

Contact			
firstName	Riccardo		
lastName	Solmi		
company	Whole Factory srl		
address	Address		
	streetAddress	Via 123	
	city	Bologna	
	country	Italy	
	postalCode	40033	
phoneNumber	PhoneNumbers		
	type	number	
	HOME	012 3456789	
	FAX	987 6543210	

Figure 2. Nested typed tables (Generic Table notation)

Mind maps are a clear example that the amount of specificity that can be provided by their notation greatly exceeds the structural constraints imposed by a specific metamodel. Furthermore, the latter can be reintroduced via tooling.

We also noted that a generic notation with just small graphical adjustments is enough to convey a domain specific feeling. Especially if the domain does not have its own well recognized specific notation.

See in Figure 1 an example of contact information written with the JSON [12] specific metamodel and notation (that are both generic with respect to the `Contacts` domain). See also the same example written in Figure 2 with a specific `Contacts` language and showed with the generic `Table` notation of the Whole Platform.

In both examples, the notation feels specific enough and the additional specificity provided by the use of a content specific metamodel in Figure 2 is mainly noticeable in user interactions.

These considerations have led us to define the `Entities` language as described in the remaining subsections.

3.1 Generic Virtual Entities

The `Entities` metamodel defines two complementary groups of entities: one for modeling an arbitrary instance and the other for modeling the corresponding metamodel. The

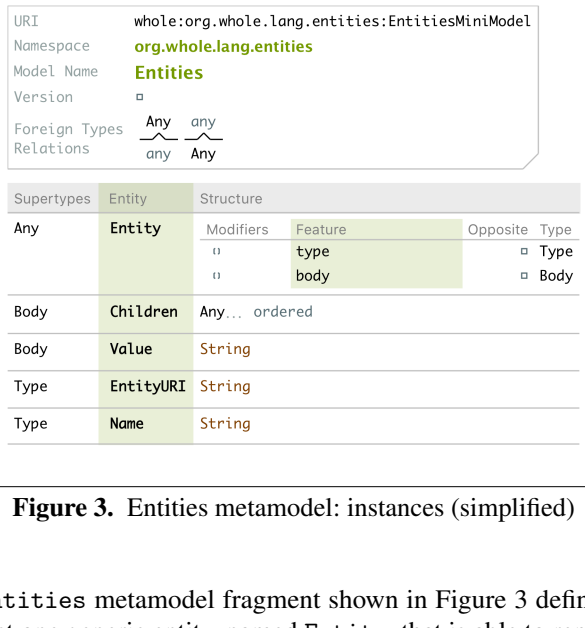

Figure 3. Entities metamodel: instances (simplified)

`Entities` metamodel fragment shown in Figure 3 defines just one generic entity, named `Entity`, that is able to represent the content and type information of any model instance. The `body` contains either a value or a recursively defined collection of children entities. The foreign type relations enable an `Entity` to be attached to any foreign language entity and, in turn, any foreign entity can become a child of `Entity`. The `Type` of an instance is either the URI of a foreign entity type or an explicit or implicit type name; implicit types are used by default to overcome missing type name information.

The goal of representing the structure of any instance model can also be reached in other ways. For instance the `JSON` [12] data format language is able to represent the field names of a record structure at the instance level without the need of any type information. `Entities` in order to discriminate a record structure from a collection needs to use the type information and stores the field names in the type declarations.

To better understand the structure of an instance of `Entities` we used in Figure 4 an AST like notation to visualize an example of a specific instance of the `Contacts` metamodel and the corresponding generic instance encoded with `Entities`.

The `Entities` metamodel fragment shown in Figure 5 defines a `MetaScope` that associates a model instance to the meta declarations used to define its metamodel. The entity types include values, collections and records. An additional union type is used to support polymorphic type hierarchies in a way suitable for an implicit, instance driven, definition of the metamodel.

The goal of achieving a reasonable level of domain specificity, despite the generic structure, is met by adding a notation feature in the `EntityTypes`. Typically, generic metamodels, such as the ones of mindmapping languages, add the notation properties directly to the instance level and avoid introducing any type information.

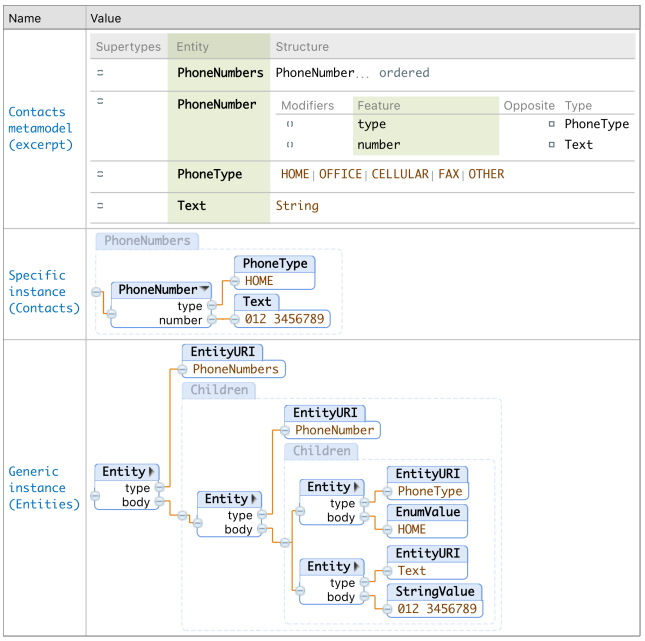

Figure 4. Specific vs generic AST of instances

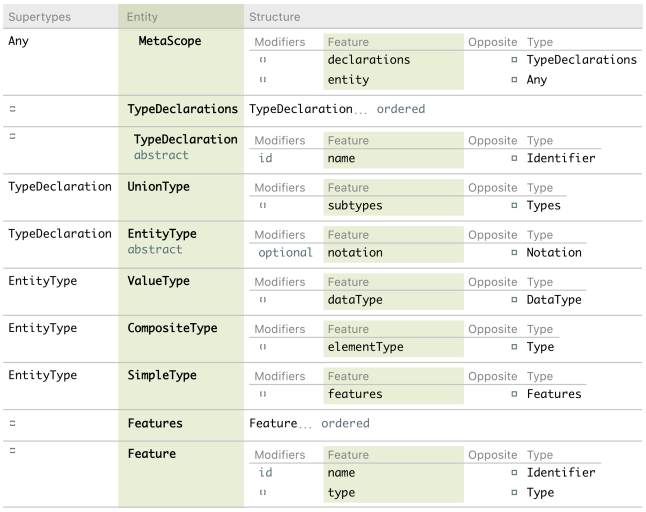

Figure 5. Entities metamodel: types (excerpt)

Our choice of defining a metamodel even if we are focused on just one instance model reflects our additional goal of minimizing the differences with the regular definition of a domain specific language.

3.2 Generic Virtual Notation

An additional customizable generic notation has been designed starting from the existing `Table` and `Tree` generic notations of the Whole Platform. The customization is performed by a declarative styling language.

The specific notation defined for the `Entities` language is the same customizable generic notation available for every

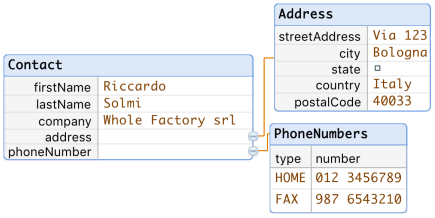

Figure 6. Tree with a composite table

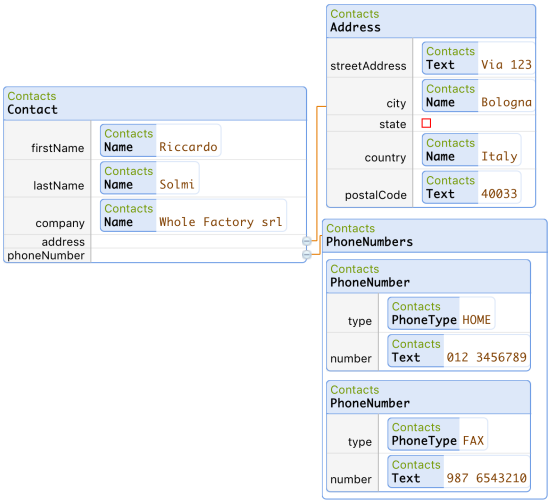

Figure 7. Full typed tree with a composite list

language, just applied to the virtual entities. In this way the transformation of a model fragment from its specific representation to the generic one, based on `Entities`, or viceversa, keeps the same familiar notation.

Each entity can choose the visibility of a header containing its type information. In Figure 1 the headers are all hidden; in Figure 2 and Figure 6 just `Contact`, `Address`, and `PhoneNumbers` are shown. The green language qualifiers shown in Figure 7 represent existing types of foreign languages.

Each entity has an embedding style that can make it appear as a standalone node (see `Contact`, `Address`, and `PhoneNumbers` in Figure 6) or as a content that blends with its context (all entities in Figure 1 and Figure 2).

Each `feature` of a `SimpleTyped` entity can be either hidden or shown inside of the corresponding cell (see `firstName`, `lastName`, and `company` in all example figures) or in a separate branch (see `address`, `phoneNumber` in Figure 6 and Figure 7).

A `CompositeTyped` entity (such as `PhoneNumbers`) can display its children either in a table (see Figure 2 and Figure 6) or in a list (see Figure 1 and Figure 7). Finally, also the visibility of table headers (light gray background color in the figures) can be changed.

3.3 Generic Virtual Tooling

The structure of the instance level, when modeled using the `Entities` language, can be regarded as a virtualization of the underling modeling framework to the domain level as seen in Figure 4. We are able to specify the same structural constraints but they are no longer enforced by the framework, rather, they need to be enforced or validated at the domain level. The tooling for the `Entities` language has been designed to hide the presence of an instance specific metamodel and to let the user concentrate himself on the instance modeling activities.

We think that the tooling should be able to operate with three different, user selectable, strategies: learning, recovering, and enforcing. In the learning mode, the modeling activities are unrestricted, the instance is assumed to be valid and the metamodel is automatically restructured in order to be able to validate the instance. In the recovering mode, instance modeling is still unrestricted, but the metamodel is used to validate the instance and any inconsistency is annotated in the model in order to help the user solve problems. In the enforcing mode, the instance modeling activities are restricted to those allowed by the metamodel in order to keep the instance valid.

4. Additional Use Cases

4.1 Dealing with Meta Errors

When a persistence stream is deserialized into a model instance, each entity description is mapped to an entity instance of the declared language. A deserialization exception occurs whenever a referenced language or entity is not available or an instance description does not conform to the definition available. The `Entities` language can be used as a fine grained replacement for the entities that are not deserializable. The resulting model is complete: it makes use of specific language entities wherever possible, and uses adaptive entities of the `Entities` language to recover from exceptions.

In case of a missing language or language version, the overall model is still viewable, editable and can be saved back without loss of information. Of course, the notation and the modeling services available on the adaptive entities are limited.

The user may perform recovering activities both at instance and meta levels. For instance, if a missing language becomes available or is defined, the adaptive entities can be replaced back to the corresponding specific entities even at runtime. In the same way, the user may fix, interactively in the editor, the errors in the instances in order to regain conformity with their declared metamodels.

4.2 Explorable Evolution

The instance modeling activities are routed and bound by existing metamodels and, in general, this is desirable. When a user feels that a given instance is not able to properly

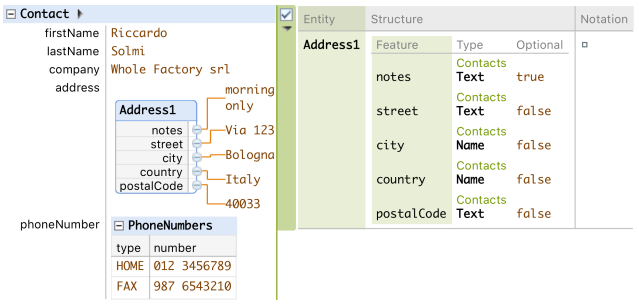

Figure 8. Instance driven evolution of Contacts

incorporate a piece of information, it can reshape the instance with an adaptive entity to overcome either a structural or a compositional limitation of the existing metamodels. In order to make the experience as smooth as possible, it is important that the transformation that replaces an arbitrary specific fragment with the corresponding adaptive fragment of the `Entities` language is also able to translate the specific metamodel information into a set of meta constraints for the resulting adaptive fragment. In this way, the only perceived difference between the original specific entity and the new adaptive entity is the desired adaptation introduced by the user. From this point on, the evolution of the instance model is no longer bound by the original metamodels and may continue by means of the facilities provided for standalone instance modeling. In Figure 8 there is an instance of the `Contacts` language in which the specific `Address` entity has been replaced by an adaptive entity, named `Address1`, with an additional feature (`notes`), a removed feature (`state`) and a renamed feature (`street`). Note that the meta data shown in the right part of the Figure 8 can be derived from the user interactions made on the model shown in the left part of the same figure.

5. Conclusions

Although the `Entities` language has been in development for several years, it is only in recent months that we were able to understand the changes needed to make the solution viable. First, we changed the `Entities` metamodel in order to move field names and styling information from the instances to the meta level. Secondly, we introduced a new notation that combines our earlier generic notations, based on tables and trees, by means of a customizable style.

The styled notation, even alone, allows to achieve reasonable levels of specificity for many domains including data formats and knowledge organization and visualization.

Standalone instance modeling is really promising, not only does it add flexibility, but it also extends the applicability of domain specific modeling to people and domains that once were difficult to reach.

References

[1] Riccardo Solmi. *Whole Platform*. PhD thesis, University of Bologna, March 2005. UBLCS 2005-07.

[2] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015.

[3] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Towards live language development. In *Proceedings of the 2nd International Workshop on Live Programming*, LIVE '16, 2016.

[4] Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S Kolovos, and Richard F Paige. Type inference using concrete syntax properties in flexible model-driven engineering. *Flexible Model Driven Engineering Proceedings (FlexMDE 2015)*, page 22, 2015.

[5] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. Example-driven meta-model development. *Softw. Syst. Model.*, 14(4):1323–1347, October 2015.

[6] Athanasios Zolotas, Dimitris S Kolovos, Nicholas Drivalos Matragkas, and Richard F Paige. Assigning semantics to graphical concrete syntaxes. *XM@ MoDELS*, 1239:12–21, 2014.

[7] Bastian Roth, Matthias Jahn, and Stefan Jablonski. Rapid design of meta models. *International Journal on Advances in Software*, 7:31–43, 2014.

[8] Bastian Roth, Matthias Jahn, and Stefan Jablonski. On the way of bottom-up designing textual domain-specific modelling languages. In *Proceedings of the 2013 ACM Workshop on Domain-specific Modeling*, DSM '13, pages 51–56, New York, NY, USA, 2013. ACM.

[9] Kacper Bak, Dina Zayan, Krzysztof Czarnecki, MichałAntkiewicz, Zinovy Diskin, Andrzej Wasowski, and Derek Rayside. Example-driven modeling: Model = abstractions + examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1273–1276, Piscataway, NJ, USA, 2013. IEEE Press.

[10] Jesús Sánchez-Cuadrado, Juan De Lara, and Esther Guerra. Bottom-up meta-modelling: An interactive approach. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 3–19, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Hyun Cho, Jeff Gray, and Eugene Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, MiSE '12, pages 22–28, Piscataway, NJ, USA, 2012. IEEE Press.

[12] *The JSON Data Interchange Format*. ECMA International, first edition edition, October 2013.