

Instance Modeling Assisted by an Optional Meta Level

Riccardo Solmi

Whole Factory srl

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 levels 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]: 6

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

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 speedup 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 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 do 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 eventually we outline other usage scenarios.

2. DSM 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

[Copyright notice will appear here once 'preprint' option is removed.]

others can be derived, postponed, or omitted by relying on generic implementations.

The language designer just 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 support for interpretation or dynamic compilation 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 speed improvement 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 entities 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 languages.

Mind maps are a clear example that the amount of specificity that can be provided by the notation exceeds greatly 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.

firstName	Riccardo
lastName	Solmi
company	Whole Factory srl
address	Via 123 Bologna Italy 40033
phoneNumbers	type number HOME 012 3456789 FAX 987 6543210

Figure 1. Nested tables (JSON specific notation)

Contact	firstName	Riccardo
	lastName	Solmi
	company	Whole Factory srl
	address	Via 123 Bologna Italy 40033
	phoneNumbers	type number HOME 012 3456789 FAX 987 6543210

Figure 2. Nested typed tables (Generic Table notation)

See in Figure 1 an example of contact information written with the JSON [12] specific metamodel and notation (that are generic with respect of 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 Entities metamodel fragment shown in Figure 3 defines just one generic Entity 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 types 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.

URI	whole:org.whole.lang.entities:EntitiesMiniModel												
Namespace	org.whole.lang.entities												
Model Name	Entities												
Version	0												
Foreign Types	Any any												
Relations	any Any												
Supertypes	Entity												
Entity	Structure												
Any	<table border="1"> <tr> <td>Entity</td> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ type</td> <td> □ declarations</td> <td> □ TypeDeclarations</td> <td></td> </tr> <tr> <td> □ body</td> <td> □ entity</td> <td> □ Any</td> <td></td> </tr> </table>	Entity	Modifiers	Feature	Opposite Type	□ type	□ declarations	□ TypeDeclarations		□ body	□ entity	□ Any	
Entity	Modifiers	Feature	Opposite Type										
□ type	□ declarations	□ TypeDeclarations											
□ body	□ entity	□ Any											
Body	Children Any... ordered												
Body	Value String												
Type	EntityURI String												
Type	Identifier abstract												
Identifier	Name String												
Identifier	Nameless int												

Figure 3. Entities metamodel: instances (simplified)

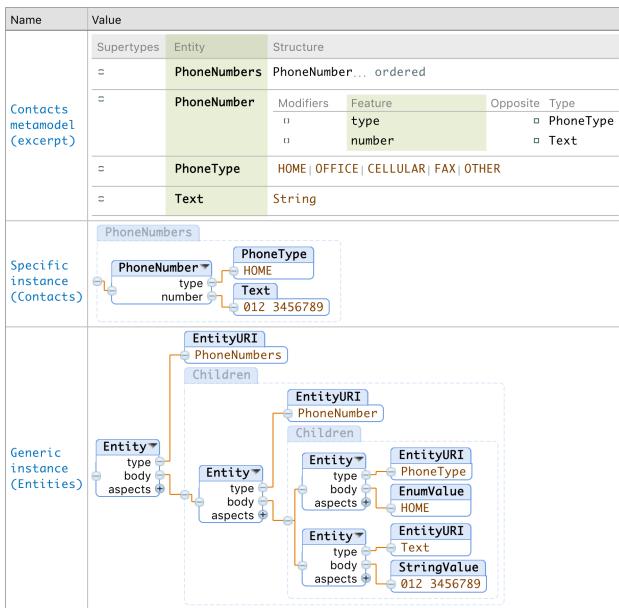


Figure 4. Specific vs generic AST of instances

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 Concepts metamodel and the corresponding generic instance encoded with Entities.

Supertypes	Entity	Structure									
Any	MetaScope	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ declarations</td> <td></td> <td>□ TypeDeclarations</td> </tr> <tr> <td> □ entity</td> <td></td> <td>□ Any</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ declarations		□ TypeDeclarations	□ entity		□ Any
Modifiers	Feature	Opposite Type									
□ declarations		□ TypeDeclarations									
□ entity		□ Any									
□	TypeDeclarations	TypeDeclaration... ordered									
□	TypeDeclaration abstract	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ id</td> <td> □ name</td> <td>□ Identifier</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ id	□ name	□ Identifier			
Modifiers	Feature	Opposite Type									
□ id	□ name	□ Identifier									
TypeDeclaration	UnionType	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ subtypes</td> <td></td> <td>□ Types</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ subtypes		□ Types			
Modifiers	Feature	Opposite Type									
□ subtypes		□ Types									
TypeDeclaration	EntityType abstract	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ optional</td> <td> □ notation</td> <td>□ Notation</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ optional	□ notation	□ Notation			
Modifiers	Feature	Opposite Type									
□ optional	□ notation	□ Notation									
EntityType	ValueType	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ dataType</td> <td></td> <td>□ DataType</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ dataType		□ DataType			
Modifiers	Feature	Opposite Type									
□ dataType		□ DataType									
EntityType	CompositeType	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ elementType</td> <td></td> <td>□ Type</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ elementType		□ Type			
Modifiers	Feature	Opposite Type									
□ elementType		□ Type									
EntityType	SimpleType	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ features</td> <td></td> <td>□ Features</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ features		□ Features			
Modifiers	Feature	Opposite Type									
□ features		□ Features									
□	Features	Feature... ordered									
□	Feature	<table border="1"> <tr> <td>Modifiers</td> <td>Feature</td> <td>Opposite Type</td> </tr> <tr> <td> □ id</td> <td> □ name</td> <td>□ Identifier</td> </tr> <tr> <td> □ type</td> <td></td> <td>□ Type</td> </tr> </table>	Modifiers	Feature	Opposite Type	□ id	□ name	□ Identifier	□ type		□ Type
Modifiers	Feature	Opposite Type									
□ id	□ name	□ Identifier									
□ type		□ Type									

Figure 5. Entities metamodel: types (excerpt)

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 to introduce any type information.

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 language, just applied to the virtual entities. 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 an header containing its type information. In Figure 1 they 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.

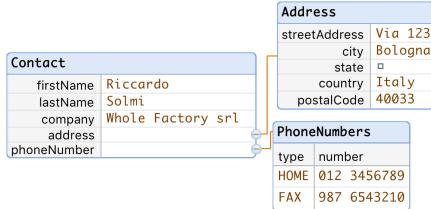


Figure 6. Tree with a composite table

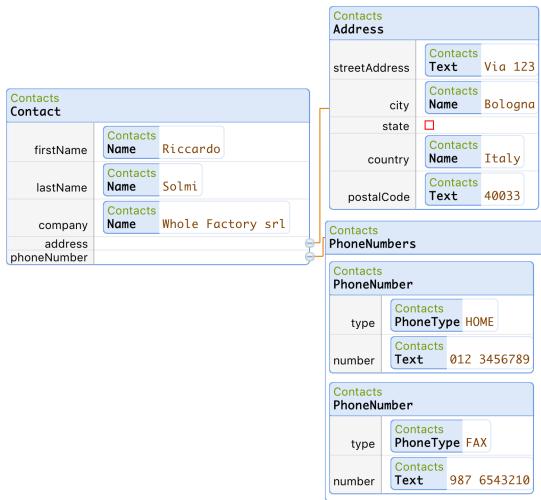


Figure 7. Full typed tree with a composite list

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, 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 meta-

model and to let the user concentrate itself 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 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 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 the problems. In 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 Recovering 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 not deserializable. The resulting model is complete, lossless and make use of specific languages wherever possible. In a separate recovering phase, that may happen interactively in the editor, the user may fix the errors.

4.2 Embedding language definitions

The Entities language provides a MetaScope entity hosting additional meta declarations that are available in the scope. By using this facility, it is possible to define a model that introduces new abstractions and make use of them. Furthermore, the editor is able to show and manipulate the new abstractions using the specific structure and notation introduced by the meta declarations.

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 viable the solution. First, we changed the Entities metamodel in order to move field names and styling information from the instances to the meta level. Secondly, we have 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 adds flexibility, but it also extends the applicability of domain specific modeling to people and domains difficult to reach before.

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, Clemens 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@ MoEWS*, 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, Michał Antkiewicz, 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.