Breaking Free of the Implementation Through Explicit Type/Instance Relations

Yentl Van Tendeloo
University of Antwerp
Yentl.VanTendeloo@uantwerpen.be

Hans Vangheluwe
University of Antwerp and McGill University
Hans.Vangheluwe@uantwerpen.be

Abstract

One of the shortcoming of current (meta-)modelling tools is their strong reliance on their implementation level. While it does offer its benefits, certainly for tool developers, it seriously impedes portability of models. Model management operations are handcoded in the implementation language of the tool, making it difficult for users to grasp their semantics. Furthermore, model management operations themselves have strong reliance on the internal data structures used by the tool, making comparison of algorithms, even at a conceptual level, difficult. In this paper, we analyze the reasons and effects for this strong reliance on the implementation level. We offer a solution which allows the explicit modelling of model management operations in a strict metamodelling framework. Even those operations that require access to multiple levels in the modelling hierarchy are supported. To aid in this effort, we furthermore break the strong link between model management operations and the data structures in use by the (meta-)modelling tool. Our technique is illustrated through the explicit modelling of a retyping operation on a petri net.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

One of the shortcomings of current (meta-)modelling tools is their strong reliance on their implementation level. This reliance ranges from exposing the general purpose implementation language used (*e.g.*, Java), to requiring some operations to operate directly on the internal data representation of the models (*e.g.*, XMI). Such strong reliance on the implementation level offers some benefits though, such as higher efficiency both in execution (as it is likely compiled) and development (tool developers are familiar with the programming language).

The disadvantages, however, are significant: models and algorithms become highly specific to the current state of the implementation, making it impossible, or at least difficult, to port models and algorithms from one tool to the other. These disadvantages not only prevent porting models between tools, but models can also become incompatible with newer versions of the tool. Should, for example, the internal data representation be changed ever so slightly, all previously created models become incompatible. Similarly, if the tool is ported from one implementation language to the other (*e.g.*, from Python to C, for efficiency), all fragments of Python code in all models would have to be updated to C code too. Even more importantly for the scientific community, is the portability of algorithms. Every year, new algorithms are introduced to further the state of the art in model management operations. These algorithms, however, often strongly rely on the internal representation of models, such as XMI or graphs. Implementing these algorithms on a tool with a different internal representation could prove challenging due to the different assumptions that can be made. Similarly, these algorithms are implemented in the implementation language of the tool, making a reimplementation of these algorithms in a tool with a different implementation language a non-trivial task as it is. Yet another disadvantage relates to tool semantics: each tool has its own interpretation of what it means to instantiate a model and check its conformance. While the semantics is obvious to experienced users, switching between tools will frequently necessitate lookups in tool documentation to understand the semantics. Related to this, porting a model between tools also requires adaptation to this changed semantics, making it even more difficult than it already is. Tool semantics are therefore non-obvious and, more problematicly, hardcoded inside of the tool. So in order to completely understand the semantics, it becomes necessary to read the source code of the tool, which is a completely seperate entity from the tool itself (*i.e.*, requires a

different viewer/editor, is in a different language, and is at a completely different level of abstraction).

To counter these disadvantages, we intend to break the strong reliance on the implementation level of tools. First, we observe why tools voluntarily chose for this strong reliance. One of the main reasons is undoubtedly purely pragmatic: tool developers are (very) familiar with specific programming languages, and thus use it wherever they can. As general purpose programming languages are a well-developed field, advanced tools are available, such as, efficient compilers, debuggers, and code analyzers. Additionally, many libraries are available for use, such as graphical libraries, parsers, data structures, and so on. This results, at least at first, in efficient code and fast development of new tools.

But even adventurous tool developers, who want to model as much as possible, quickly hit a wall: the complete system, thus including the model management algorithms, needs to become independent of the implementation language, and should thus be explicitly modelled. It is these algorithms, however, that impose constraints such as conformance and strict metamodelling. And while it is possible to bootstrap these algorithms, the model of these algorithms will, by definition, violate strict metamodelling requirements. A simple example is the conformance algorithm: to determine whether a model conforms to another, information is required from both the metamodel and the model, thus combining two modelling layers. This is the exact thing that strict metamodelling prevents, as it prevents models from spanning multiple levels.

This problem is shown in Figure 1, where a petri net model linguistically conforms to a simple petri net metamodel. A conformance algorithm, however, has to access parts of the metamodel (*e.g.*, *Place*) and parts of the model (*e.g.*, *a place*), to check whether they conform. This makes it difficult to state on which level the algorithm itself has to reside: at the model or metamodel level. In the figure, this is shown through the use of a gradient: it sits somewhere in between the levels. Note, however, that each element does indeed conform to the physical level in a strict way. The physical level is part of the implementation and is therefore not (meant to be) user-accessible.

The natural solution to this problem, as followed by most tools, and in particular deep metamodelling tools, is to shift these strict metamodelling violating operations to the physical conformance dimension. In that dimension, all models, even metamodels, become part of a single level: the implementation level. As the implementation level is implemented in the implementation language, no restrictions are imposed whatsoever, as models and metamodels are both just elements in the data structure.

While this approach has served current (meta-)modelling tools well, it prevents the explicit modelling of the complete system, resulting in the strong reliance on the implementation level, with all previously associated problems. This problem is aggrevated in deep metamodelling, as there can be absolutely no assumptions in the linguistic dimension: users can create an arbitrary number of layers. Deep metamodelling further raises problems related to strict metamodelling, such as how to specify deep constraints (Atkinson et al. 2015), which also span multiple levels. While we would prefer to model these explicitly, this becomes impossible due to the explicit level-crossing nature of the algorithms.

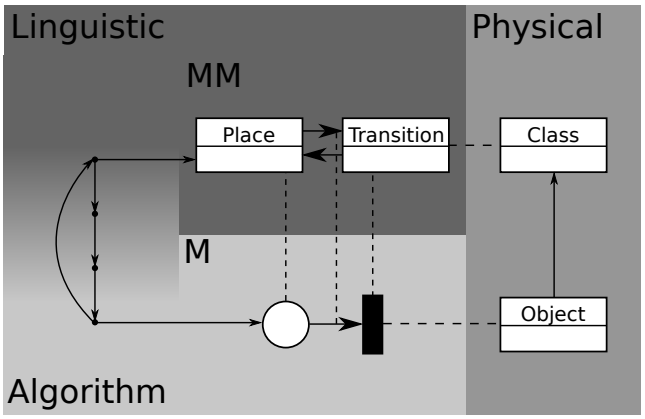


Figure 1: Illustration of the problem: the algorithm spans both the metamodel (M2) and model (M1) layer in the linguistic dimension, thus violating strict metamodelling.

In this paper, therefore, we plan to tackle this problem of strict metamodelling, allowing models of model management operations, while still keeping strict metamodelling in its original meaning. We do this by shifting parts of the physical conformance level, normally hardcoded in the tool, to the linguistic conformance level, where users can access it just like any other model. Not only does this allow for linguistic modelling of tool algorithms, but it decouples these algorithms from implementation details.

Apart from explicitly modelling the tool itself, we believe that this approach serves well in combination with megamodelling (Bézivin et al. 2005), where we start reasoning about inter-model relations. Currently, most megamodel management operations are still implemented in implementation languages, such as Java (Salay et al. 2015). Certainly the combination with runtime models (Vogel et al. 2010) has the potential to highly profit from our approach.

The remainder of this paper is organized as follows. Section 2 summarizes the required background in supporting explicitly modelled type/instance relations, and how multiple conformance relations are enabled through the application of this technique. Section 3 touches upon the three main dimensions in terms of conformance, and presents how we encode each of these relations. Section 4 presents our approach to merging the physical conformance level into the linguistic conformance level through the use of $conformance_{\perp}$. Re-

lated work is explored in Section 6. Section 7 concludes the paper and gives future work.

2. Background

In this section, we briefly introduce the required concepts for the remainder of the paper. As our contribution is focussed on moving away from the implementation layer, we present our previous work in doing so. First, we discuss our explicitly modelled, neutral action language. It is a simple procedural action language, but its constructs are explicitly stored as a model themself (*i.e.*, an abstract syntax graph). Second, we use this action language to explicitly model the type/instance relations, showing the benefits we gain from doing so. This explicit type/instance relation offers us multiple possible typings for a single model, which sits at the core of our approach.

Each part will already briefly encounter the problems mentioned earlier in this paper. Solutions to this were always rather ad-hoc, as our primary intention was on bootstrapping our tool: the Modelverse (Van Mierlo et al. 2014). After the implementation of the enabling technology, presented in this section, we have come to an elegant solution, presented in this paper, to a recurring problem.

2.1 Explicitly Modelled Action Language

A first step in breaking free from the implementation layer, is by removing all code snippets in the implementation language that are scattered throughout the model. Since we don't want to depend on a single implementation language, a neutral action language was defined.

While our action language serves a similar purpose as other neutral languages, like the OCL, its major difference is that it is explicitly modelled. Whereas an OCL interpreter follows the OCL standard, a 262-page specification, issued by the OMG (OCL 2014), our action language is formalized at a much lower level. Only a handful of low-level operations are supported, which are very similar to the primitive operations in any procedural language. As we store models as graphs internally, action language models are also represented as a graph. Semantics are formally defined through the use of graph transformations, defining a translation from one graph pattern (before execution of the instruction) to another (after execution of the instruction). Our interpreter itself is therefore only a few hundred lines of code, to which a set of graph transformation rules are passed. Only these few hundred lines of code are dependent on the underlying platform, and should be ported when porting the complete application.

Apart from explicitly modelling the semantics, this also makes sure that the execution state is explicitly modelled, as that too is modified by the graph transformation. The major disadvantage, however, is performance, which is very low at the moment.

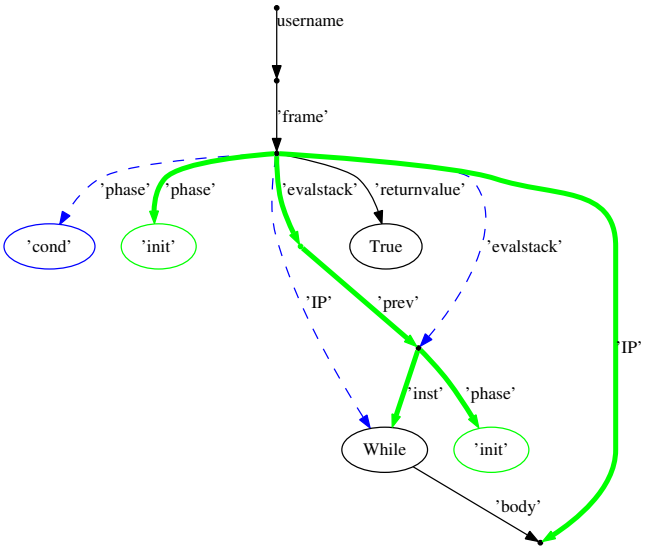


Figure 2: Rule to execute when a `while` is executed with a condition evaluating to `True`.

An example rule is shown in Figure 2, where the graph transformation rule is shown for when a `while` element is to be executed with a condition that evaluates to `True`. Note that not only the action language constructs are explicitly modelled, but also the complete execution context (*i.e.*, execution stack) is explicitly represented in the Modelverse and is transformed according to these rules.

It is our intention to create a library of operations, containing, for example, the operations defined by OCL, which is modeled using this minimal action language. But while this is a noble plan, we quickly encounter the same problems that were raised in the beginning of this paper: strict metamodelling does not allow for models that span multiple levels. An explicit model of, let's say, a conformance relation, needs access to both the type and instance level, as shown in Figure 1. We temporarily solved this problem by ignoring strict metamodelling requirements, such that we can bootstrap the tool anyhow.

The contribution of a new action language itself, is therefore not sufficient to address the problems raised at the start of this paper. While some code snippets in models can be replaced with implementation-independent action code, such as OCL or even our own action language, this action language is not sufficient to model low-level operations that need to work across multiple levels in the modelling hierarchy. Model management operations, in particular, frequently require cross-level access to the model.

2.2 Explicitly Modelled Type/Instance Relations

A next step in allowing for our contribution, is to explicitly model the type/instance relation between models. By explicitly modelling the type/instance relation, we have shown that users gain more insight in tool semantics and can alter the

semantics if desired. This resulted in the possibility for multiple types of type/instance relations. For example, a single model can be typed by multiple metamodels, possibly simply because they are different, but similar, metamodels, but maybe also because a less restrictive conformance check is being used. As there is no common agreement on how restricted a conformance relation should be (*e.g.*, should it take into account all of potency, cardinality, or multiplicity, and how should it behave if these are violated?)

We achieved this explicit semantics through explicit modelling of the type/instance relation, which constisted of several steps.

First, the metamodel was stripped to only contain structural restrictions. As such, there were no longer any additional attributes that were undefined at the level above, such as potency, multiplicity, cardinalities, and so on. Attributes with that name could be present though, if they were allowed by the metamodel, but they have no associated semantics, thus not restricting instances.

Second, type information from the model was split of into a seperate model. A model was thus reduced to a mere graph, which structurally conformed to another graph (the metamodel). The type information contained links between both graphs, indicating the type of each element in the model. Due to this seperation, a model could possibly have multiple type mappings, together with multiple metamodels. To determine whether a model was typed by another model, both models were required, but additionaly a mapping also needed to be present.

Third, all semantics was shifted to the instantiation and conformance checking algorithms, which make up the type-/instance relation. The instantiation algorithm checks all necessary constraints during instantiation, and, for example, prevents further instantiation if the potency has already reached zero. Similarly, the conformance check checks both the structure, as defined by the graphs, the types, as defined by the type mapping, and the additional constraints imposed by giving semantics to special attributes like potency and cardinalities. It is thus now the conformance algorithm which gives the semantics to these attributes, and no longer the tool internals, which are hidden from the user and non-modifiable. To make the algorithms independent of the implementation, they are implemented using the previously defined neutral action language, which is explicitly modelled.

The most important aspect, in the context of this paper, is that it allows a single model to conform to multiple metamodels simultaneously, possibly even through different conformance checking algorithms. This is shown in Figure 3, where a single petri net model is shown (as a graph; in the middle) which conforms to two different metamodels: an ordinary place/transition net metamodel (top), and one extended with an inhibitor arc (bottom). Additionally, minor differences exist between the two metamodels, such as different naming for the *weight* of the transition. When a users uses the model, it is thus required to pass the corresponding type mapping, of which there are now two.

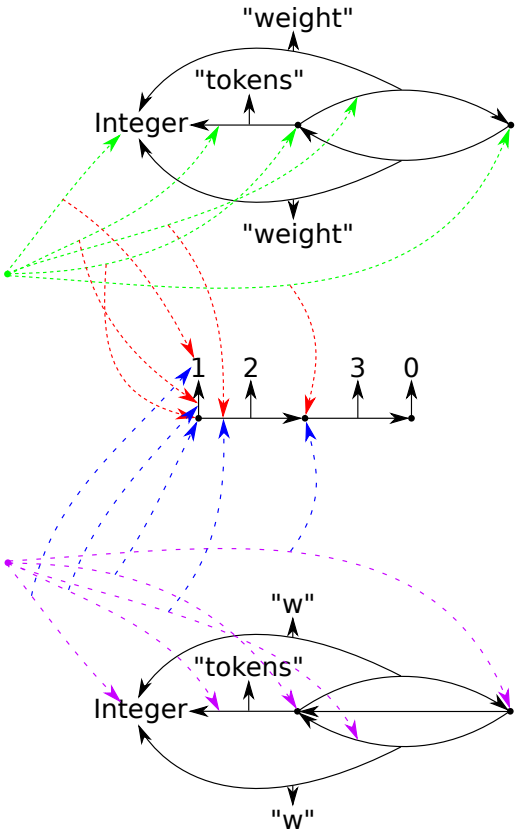


Figure 3: A single petri net model with multiple conforming metamodels, each through a different type mapping (excerpt).

Note again that we encounter the problem of strict metamodelling here. The conformance checking algorithm needs access to both the model and metamodel, thus crossing over multiple levels in the modelling hierarchy. This problem was again alleviated by not caring too much about strict metamodelling, thus again moving away from normal modelling, and into a realm between the tool implementation and the actual models in the tool. In this paper, we finally shift all parts from the implementation to the models in the tool.

3. Types of Conformance

It was found out that there is not a single kind of type/instance relation, but actually multiple, to cope with the different kinds of relations (Atkinson and Kühne 2003). A distinction was made between conformance to the physical implementation, and to the linguistic metamodel. This classification architecture was termed the Orthogonal Classification Architecture (OCA). Other work (Barroca et al. 2014; Vanherpen et al. 2016) has shown the presence of yet another type/instance relation, which links back to the semantics of

the model. Depending on which dimension is used, different type/instance hierarchies emerge, changing the implications of strict metamodelling. Sadly, terminology is rather inconsistent in the literature, making it difficult to clearly communicate about the different dimensions. This section serves to clear up the terminology we will use throughout this paper.

A concise example is shown of a simple petri net model, which conforms to three different metamodels with respect to these different conformance relations.

3.1 Physical Conformance

The low-level view on conformance is what we call *physical conformance*. This is the kind of conformance used by tool-developers and model management operations, as it does not impose many constraints. It sits at the physical level, where it is responsible for how data is represented physically in memory. Physical representation is unrelated to language engineering, and is completely managed by the implementation language of the tool. Physical conformance, therefore, is checked by their respective compilers (*e.g.*, GCC and Java compiler).

But since it is checked only by the compiler, no restrictions are placed upon the crossing of modelling levels. Indeed, all models, metamodels, metametamodels, and so on, are similar at this level: they are data created by the user that can be manipulated by the tool. Tool internals, as well as model management operations, are mostly implemented at this level, as it allows all possible modifications to the model. Additionally, this level is efficient due to its close relation to the implementation, and use of advanced tools such as efficient compilers. Since this is the lowest level of conformance, to which every model conforms by definition, all operations need to be applicable to all models. Most of the time, the only common representation of all models is the data structure used to store them. These operations thus alter the internal data structure directly, without any kind of domain-specific algorithm in between.

While these modifications are again efficient, and furthermore easy to implement for the tool developer, the strong link to the implementation is made obvious. Porting all tool internals to a different tool doesn't only require porting between programming languages, but also between different internal data representations. Changing implementation details, which should be transparent to the user, will have significant implications on these algorithms as well.

In our petri net example, this relates to how the petri net is stored in memory: with all elements being instances of a generic *Class* defined by the implementation language. This is shown in Figure 4: the model conforms to a simple graph representation, where places and transitions are represented as nodes, and the edge between them is mapped to the edge of a graph. Depending on the implementation, a different physical metamodel can be used, for example that of an SQL database. The metamodel is the same for every model, be it petri nets, class diagrams, object diagrams, statecharts,

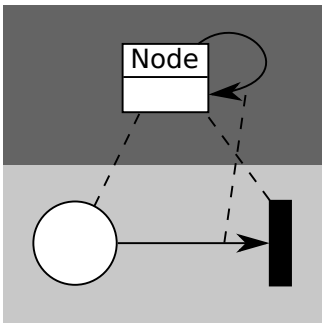


Figure 4: Physical conformance of a petri net model.

or even a domain-specific language. The reason for this is simple: if the model doesn't conform to this metamodel, the tool has no way of representing it in memory.

3.2 Linguistic Conformance

Linguistic conformance is the traditional view on conformance, which is heavily used for domain specific modelling. It is also the view offered to users: if a user creates a metamodel, and subsequently instantiates it, this is through linguistic conformance. The relation defines whether or not a user-defined model is a valid instance of another user-defined (possibly by another user) metamodel. Checks mostly have a structural notion (*e.g.*, is a link between these entities allowed and are all required attributes present), though minor semantical constraints are also possible. These semantical constraints are for example range checks on attributes (*e.g.*, integer must be larger than 5), global restrictions on values of attributes (*e.g.*, the sum of these two attributes needs to be greater than 10), or global restrictions on the structure (*e.g.*, no loop possible for a certain association). Whereas structural checks are easily implemented through the use of a metamodel, the additional semantic constraints are often implemented using some kind of executable language. Constraints are written in constraint languages, such as OCL, but are sometimes already shifted to the implementation language (*e.g.*, implemented in Python as in AToM^3 (de Lara and Vangheluwe 2002) and AToMPM (Syriani et al. 2013)).

Contrary to the physical conformance dimension, users can extend the linguistic conformance dimension by adding, modifying, or deleting metamodels. As such, a model is not, by definition, always conforming to its provided metamodel. This conformance relation is also not checked by the implementation language, but by the tool itself. But whereas most programming languages are standardized and have a clear definition on what it means to conform, this is not always the case in the linguistic dimension. Each tool has its own interpretation of what it means for models to conform to another model. Making this relation explicit, and thus offering the user the choice, was the primary effort of our previous work (Van Tendeloo and Vangheluwe 2016).

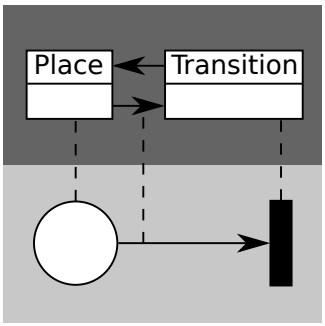


Figure 5: Linguistic conformance of a petri net model.

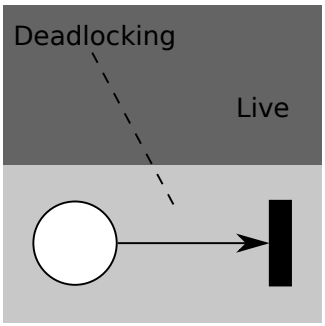


Figure 6: Ontological conformance of a petri net model.

It is also at this level that strict metamodelling comes into play: no links, except for the *instanceOf* link, is allowed to cross the levels defined by this relation. Tool users should ideally only be concerned with this dimension, as it offers support for domain-specific languages and makes use of all the features implemented by the tool (*e.g.*, strict metamodelling, type checking, model transformations, and consistency management).

In our petri net example, this relates to the metamodel of petri nets which constrains the structure of the net. Example constraints are that no direct link between places is possible (specified by the omission of an association from *Place* to itself), and the number of tokens in a place needs to be positive (specified by a static semantics constraint). This is shown in Figure 5, where each element conforms to a metamodel that is specific to the model. Contrary to physical conformance, this metamodel is only valid for petri net instances. At this level, it is impossible to create, for example, a link from the place instance to the place type, due to strict metamodelling. There is also no mention of the implementation: this relation does not imply anything on how a place is represented in memory (*e.g.*, as a node in a graph or as an ID in a SQL database). It does, however, constrain the structure of the model. The metamodel contains an association from *Place* to *Transition*, and vice versa, but no transition from *Place* or *Transition* to itself. This constrains the instances more than was the case with physical conformance. Furthermore, it is often this dimension of conformance that is used to specify concrete syntax. Strictly speaking, the representation of the model in physical conformance would have to be the actual graph that is stored in memory. We did not do this to prevent confusion.

3.3 Ontological Conformance

The final conformance dimension is *ontological conformance*, which relates purely to the semantics of the model. It is also one of the views offered to users, but is not related to the structure of the model, only to the properties the model satisfies. As it relates to semantics, execution of the model is required. Depending on the property of interest, the algorithm executed varies from, for example, simulation (*e.g.*, trace satisfies some property) to state space analysis (*e.g.*, deadlocking system).

The algorithm to be executed often again relates back to the physical dimension, as this is where the implementation is defined in case the implementation language is used.

In our petri net example, this relates to the properties satisfied by the petri net, such as deadlocking, safety, or reachability. Figure 6 represents a petri net without any tokens and not generators, so the petri net is clearly deadlocking and not live. Ontologically speaking, the petri net thus conforms to the *deadlocking* property and not to the *live* property. This concept is again broader than petri nets only, and might also be applicable to formalisms that have similar sementics or properties. But while previous conformance relations focussed purely on static aspects of the model (*i.e.*, structure and static semantics), this conformance dimension focusses exclusively on the semantics through execution.

4. Explicit Modelling of Physical Conformance

Recall that relying on the physical conformance relation was the cause of the problems we have previously observed. The theoretical limitation, preventing explicit modelling of these algorithms, were the limitations imposed by strict metamodelling: a model cannot span multiple levels. These problems let to the obfuscation of tool semantics, and the strong reliance on implementation details for all algorithms.

4.1 Moving Away from Physical Conformance

As all problems seem to be situated in the physical conformance dimension, the most direct solution would be to do away with this dimension completely. This is, however, not possible, as each model still requires a physical representation in memory, as well as model management operations defined over it.

The closest we can get, is shifting away many responsibilities of the (hidden) physical conformance dimension, into the (explicitly modelled) linguistic conformance dimension. There is a natural relation between both physical and linguistic conformance, as both are related to the structure of the model.

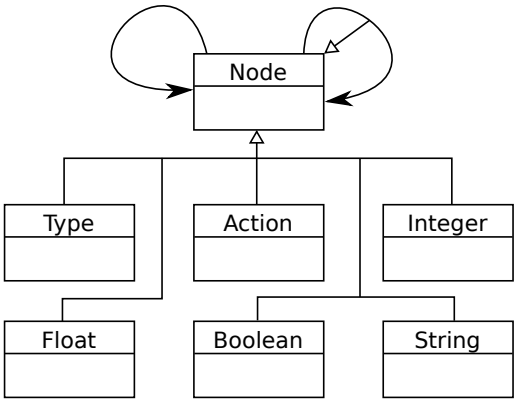


Figure 7: $LTM_{\perp}$, allowing for any element to connect to any other element.

To do this, we define a new metamodel, which is identical to the (implicit) metamodel of the implementation layer. This metamodel, however, is defined in the linguistic dimension, thus making it explicit. For clarity in our discussion, we call this metamodel $LTM_{\perp}$, shown in Figure 7. It can be seen that it is a metamodel for basic graphs, where nodes might have values. These possible values are *Type* (the type of any value type, including itself), *Action* (the type for all action language constructs, such as *While*, *If*, and *FunctionCall*.), *Integer*, *Float*, *Boolean*, and *String*. Additionaly, edges are a subclass of nodes, meaning that they can have incoming and outgoing edges themself. Since every element is a subclass of *Node*, an edge can start and end at any element, including itself. As this is only at the conceptual level, it was done to make reasoning about edges from edges conceptually clearer. The leftmost association from *Node* to itself represents the type of inheritance relations: since inheritance relations are also explicitly modelled (Van Tendeloo and Vangheluwe 2016), they require their own metamodel. And since the $LTM_{\perp}$ should be self-describing, it contains this type too.

Since any model conforms to the (often implicit) physical metamodel in the physical dimension, they should also, by definition, conform linguistically to $LTM_{\perp}$. We call this new linguistic conformance to $LTM_{\perp}$ $conformance_{\perp}$. While it is actually the same as conformance in the physical dimension, we shift this to the linguistic dimension to offer it to the users. Thanks to the possibility for multiple metamodels for a single metamodel (Van Tendeloo and Vangheluwe 2016), it is possible for the model to be typed by multiple linguistic metamodels: $LTM_{\perp}$, and the original linguistic metamodel(s). Figure 8 shows the 1-to-1 mapping of the Physical Type Model (PTM) to the linguistic dimension. As each element necessarily conforms to the PTM, it will also, by definition, conform to the new $LTM_{\perp}$.

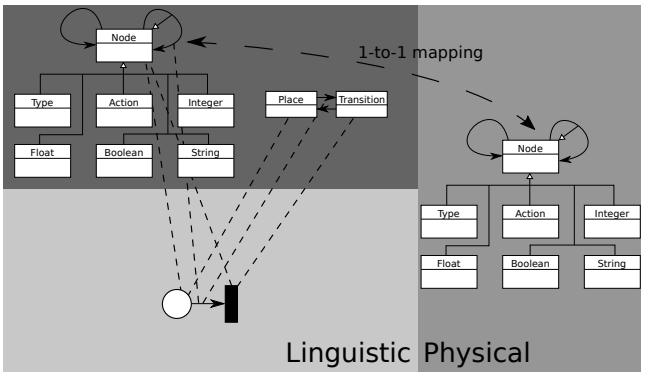


Figure 8: $LTM_{\perp}$ added in the linguistic dimension, which is identical to the one in the physical dimension.

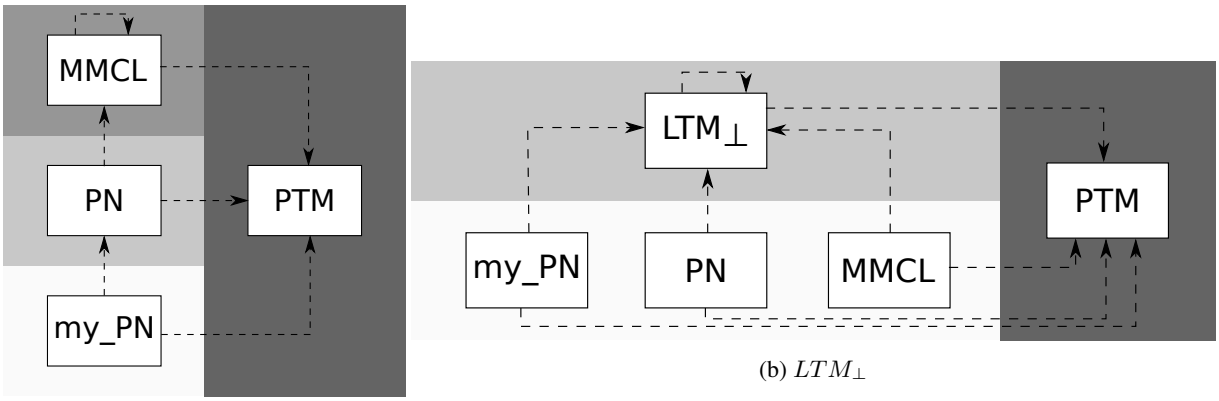
4.2 Coping with Strict Metamodelling

By lifting the physical conformance relation up to the linguistic conformance dimension, we achieve a way of explicitly modelling, albeit indirectly, in the physical dimension. Users are therefore able to, using their normal linguistic modelling tools, alter the physical dimension. The physical representation of the model is thus seen as an instance a linguistic metamodel.

While the tool still complies to strict metamodelling in the linguistic dimension, $LTM_{\perp}$ is taken so general, that the complete metamodelling hierarchy can be expressed as a direct instance of it. This effectively flattens the original metamodelling hierarchy into a single level: $LTM_{\perp}$ at the metamodelling level, and everything else at the modelling level. In this single model level, which is only a different view on the same model, strict metamodelling does not restrict anything, even links between different levels (of the original hierarchy). Figure 9 represents the two possible views on the modelling hierarchy: either through the usual conformance relation (Figure 9a), or the new $conformance_{\perp}$ relation (Figure 9b).

Depending on the used metamodel and conformance relation, strict metamodelling can thus be interpreted differently. Note that this is still distinct from dropping strict metamodelling completely: strict metamodelling is still used throughout the complete environment, and still imposed on instances, even with the $conformance_{\perp}$ relation. But the implications of strict metamodelling depend entirely on the metamodel: for normal linguistic metamodels, strict metamodelling is as it was originally designed, but for the special metamodel $LTM_{\perp}$, strict metamodelling does not constrain anything because every element is at the same level.

Coping with strict metamodelling alone does not solve all problems. While the limitation of not being able to model executable models across levels was removed, these executable models still directly interact with the underlying data structure. This is still a lingering aspect of the physical dimension, which we tackle next.



(a) Petri nets metamodel

(b) $LTM_{\perp}$

Figure 9: Different modelling hierarchies for the model *my_PN*, as seen through two different linguistic views.

4.3 Abstracting Implementation Details

The 1-to-1 mapping between the physical metamodel and $LTM_{\perp}$ made it possible to linguistically access the physical dimension. But the physical dimension is still part of the implementation, and could therefore change in subsequent versions. This would bring us to language evolution, as $LTM_{\perp}$, and possibly $conformance_{\perp}$, would also have to be updated, together with all saved models. While some advances are made to language evolution in order to do these changes automatically, we don't want to expose users to these problems.

Users should therefore not be bothered with the internals of the tool, not even the physical data representation. And while users do need access to a physical-like representation, it can certainly be a different one than that which was implemented, as long as there exists a mapping between them. $LTM_{\perp}$ is thus merely a wrapper, or an abstraction of the actual data structure being used. Modifications on instances of $LTM_{\perp}$ are mapped over to changes in the physical dimension, and vice versa. This can be done by having the actually implemented data structure implement an interface as if it were conforming to $LTM_{\perp}$. This requires a mapper between $LTM_{\perp}$ and the physical metamodel, which is similar to physical mappers (Van Mierlo et al. 2014). Now, however, the mapping is only defined for a single metamodel, instead of for each metametamodel individually, greatly relieving users. This is the mapping shown in Figure 10.

Decoupling the implementation of algorithms from the actual internal data structure makes it possible to perform drastic changes internally (*e.g.*, switching between database technologies), without any change whatsoever to the explicit models of model management operations, nor to $LTM_{\perp}$ or $conformance_{\perp}$. Related to this, different tools can implement exactly the same algorithms, which were explicitly modelled, even if their implementation language and internal data structure is completely different. They only need to agree on $LTM_{\perp}$ and the corresponding $conformance_{\perp}$, and an explicitly modelled action language to go along with it. All other implementation choices become truely that: choices made in the implementation that don't affect functionality at all.

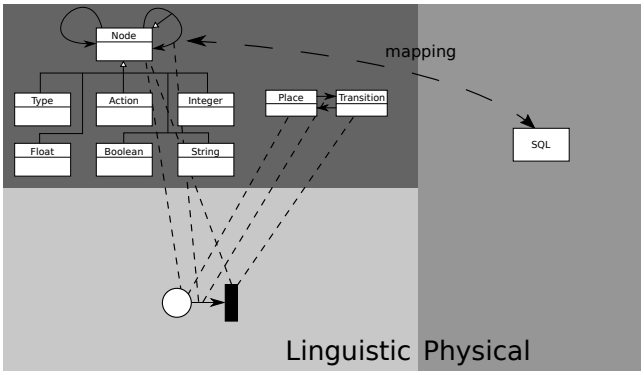


Figure 10: Changing the physical metamodel with something else, as long as there is still a mapping to $LTM_{\perp}$. SQL metamodel not expanded due to space constraints.

4.4 Overview

We now relate back to the problems we initially observed. The strong reliance on the physical dimension were caused by both pragmatic reasons (*i.e.*, developers are more familiar with programming languages) and theoretical limitations (*i.e.*, strict metamodelling prevents a model from referencing two different levels). While we can't do much about the pragmatic reasons, we have used multi-conformance to offer a different view on the model: instead of being an instance of a user-defined metamodel, it becomes an instance of $LTM_{\perp}$. Using the $conformance_{\perp}$ relation, strict metamodelling does not constrain the user anymore because all model elements reside at the same level.

Similarly, the physical implementation and mapping to $LTM_{\perp}$ were decoupled from the linguistic metamodel,

making it possible to alter the implementation without affecting $LTM_\perp$ or its instances at all.

5. Example

As a simple example of our approach, we present here the implementation of an instantiation model management operation. The operation is invoked on an element in a metamodel that has to be instantiated, using the existing model to which the instantiation should be added.

Current tools implement this using code written in the implementation language and hardcoded in the tool, even though their tool supports a neutral language (*e.g.*, OCL). A normal neutral language is unfit for this purpose for several reasons:

1. Any representable model in the tool might become subject to instantiation, so we don't want to define the operation over the linguistic metamodel defined by the user. If we were to do this, only instances of that exact metamodel could ever be instantiated. Creating a new metamodel would also require users to reimplement all instantiation operations over and over again, to make them applicable to the model used. Most of the time, instantiation is very similar, so a default should be provided. By implementing this operation at the physical level, tools avoid this problem as they now work on the internal representation, which is identical for all models.
2. Instantiating a model element is an invasive operation, which can greatly disturb the linguistic dimension by, for example, breaking conformance to the linguistic metamodel. Implementing instantiation based on the linguistic metamodel, defined by the user, would therefore also be unwise, as conformance might break halfway through the operation, making the function not applicable anymore.
3. Strict metamodelling prevents users from crossing between levels. Even if the previous two problems were to be solved, a single model (the instantiation algorithm) cannot have links to both the model (to add the instantiated element), the metamodel (to read out the element to instantiate), and even the metametamodel (to find subtyping information).

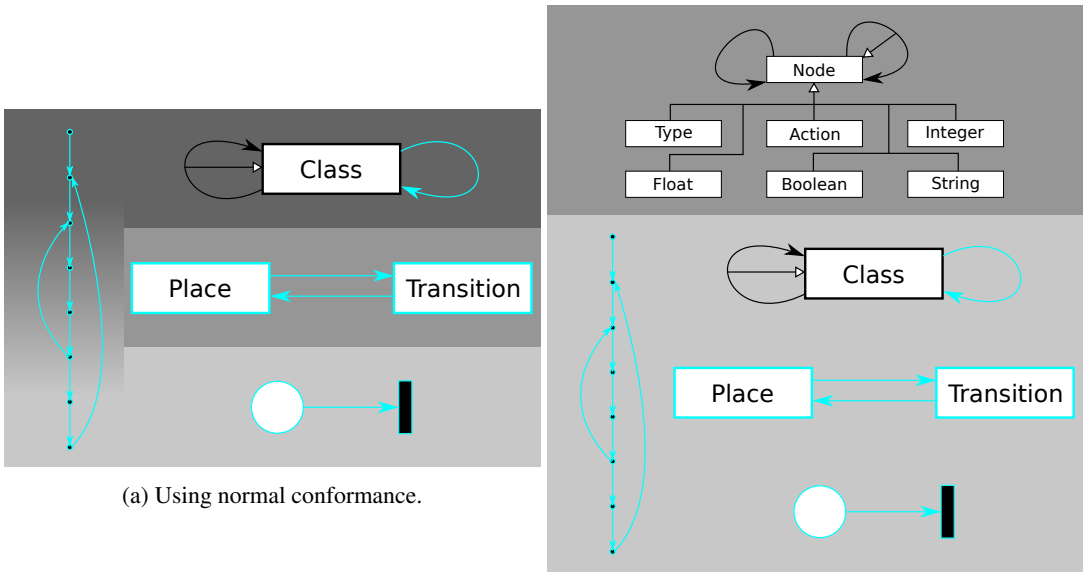
With our approach, each of these reasons is solved as follows:

1. Instead of shifting the algorithm to the physical dimension to get access to the physical representation, we shift the physical representation to the linguistic dimension as $LTM_\perp$. This way, the low-level representation of the model is also an explicit instance, for which each possible model conforms to one and the same metamodel: $LTM_\perp$. If the retyping operation is defined using $conformance_\perp$, it will be applicable to every possible model.
2. The type of a model is not visible in normal circumstances, as it is part of the conformance check. It is indeed even dangerous to change the type of a model, while operating on that specific type. By operating on a different type, however, of which it is known that the model will always conform to it, there are no risks involved at all. While it might be possible that some of the other previous conformance relations are broken (*e.g.*, to a user-defined metamodel), $conformance_\perp$ is not invalidated by the operation as it holds by definition.
3. As previously shown, our approach just changes views to $conformance_\perp$, in which strict metamodelling is still valid, but it doesn't actually restrict anything, since every element is at the same level.

The algorithm is related to how models are represented internally: all models are subgraphs of a single coherent graph. This format of model representation is itself already level-crossing, as there are edges for both navigation and instantiation. As it contains level-crossing links, it is an invalid model when viewed through an ordinary linguistic typing relation. It is, however, viewable and even modifiable using $conformance_\perp$, as the model completely complies to $LTM_\perp$.

During the execution of the algorithm, the model is viewed not through the usual conformance relation, but through the $conformance_\perp$ relation. As such, the model can be modified as if it were merely a graph, without any additional semantics or imposed restrictions. Apart from just allowing any kind of structural change, inconsistencies in the usual conformance relation are also possible: cardinalities, multiplicities, potencies, and so on, can all be invalidated as their semantics is not checked at this level. Operations defined by the user, using the normal linguistic conformance relation, will just reinterpret the graph to the usual linguistic dimension, thus again checking all additional constraints such as cardinalities.

We use this code to instantiate a new petri net place, as specified by the petri nets metamodel. The example is visualized in Figure 11. Figure 11a indicates the problem with the instantiation algorithm: it accesses itself and three different modelling levels: the model level to write out the instantiated model, the metamodel level to read out the allowed attributes and all constraints, and the metametamodel level to know about inheritance links and how to handle them. Accessed elements are highlighted in the figure, indicating that the algorithm requires access (and thus, links) to all these levels. It is therefore impossible to add it at either of these levels: adding it to one level would cause violations for the other levels. By taking the $conformance_\perp$ view, the modelling hierarchy changes from Figure 11a to Figure 11b, in which there are no level-crossing links anymore. In Figure 11b, all access are again highlighted, but are now within the same level in the modelling hierarchy. There is therefore no longer any violation of strict metamodelling.



(a) Using normal conformance.

(b) Using $conformance_{\perp}$.

Figure 11: Two different ontological views on the same model. The elements accessed by the algorithm are shown in light blue. Only $conformance_{\perp}$ complies with strict metamodelling.

The complete procedure is shown in Figure 12: first the $conformance_{\perp}$ view is taken on the model, where it is shown as a graph instead of a petri net model and metamodel. Second, this graph is traversed and the requested changes are performed. Finally, the modified graph model is again interpreted using the original conformance relation, where users use their own metamodel and corresponding type mapping to interpret the graph.

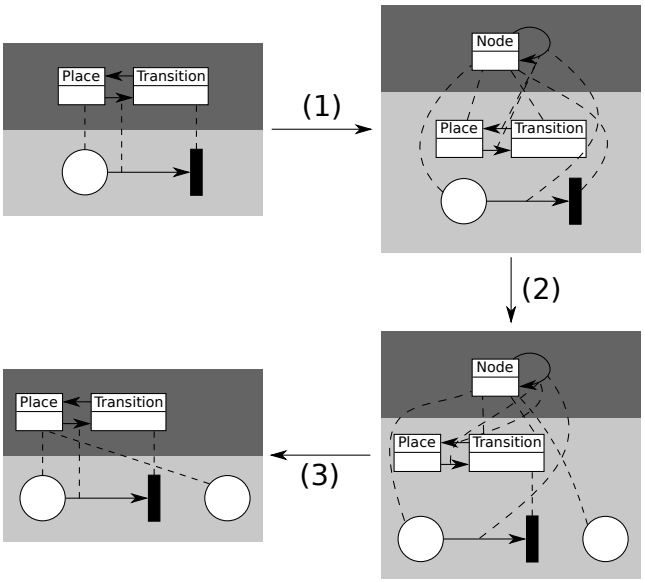


Figure 12: Overview of the complete procedure: (1) reinterpret the model as instance of $LTM_{\perp}$, (2) execute the algorithm on the graph representation, (3) reinterpret the model again using the initial metamodel. All steps happen on the background and the user only sees the composite operation.

6. Related Work

Three main dimensions of related work exist.

First, our approach builds upon the support for multiple linguistic types. While we have used our approach (Van Tendeloo and Vangheluwe 2016), another possible direction is through by a-posteriori typing (de Lara et al. 2015). In a-posteriori typing, a model is constructed with a single *constructive* type (Atkinson et al. 2011), which cannot be changed. When a model is used in a different context, however, multiple additional types can be added afterwards (*a posteriori*) through the use of concepts (de Lara and Guerra 2010b). These additional types don't influence the original constructive type, but can make the model applicable for use in other algorithms. Supporting our $conformance_{\perp}$ relation through the use of a-posteriori typing should be similar. The constructive type could simply be part of $LTM_{\perp}$, with all "real" linguistic types specified as a posteriori types. Our approach varies a bit though, since we don't make the constructive type a special kind of type: the $conformance_{\perp}$ is just another relation like any other. The OCA (Atkinson and Kühne 2003) is rather similar to our approach, as it identi-

fied the distinction between two conformance relations. But whereas the OCA shifts one of these relations to the implementation level, we merge the physical type model into the linguistic dimension. We therefore still completely comply to the OCA: we have both a linguistic dimension (used for user modelling), and a physical dimension (used during tool building). Parts of our physical dimension are, however, exposed to the linguistic dimension, such that all operations from the physical dimension also become available in the linguistic dimension. With the OCA it is not necessary to support multiple linguistic types for a single model, which is a necessary requirement when shifting more parts to the linguistic dimension.

Second, strict metamodelling has been the subject of several debates, both in favor (Atkinson and Kühne 2001, 2005), and against (Henderson-Sellers et al. 2013; Clark et al. 2014). People against strict metamodelling argue that strict metamodelling makes specific models impossible, as we have also shown in this paper. Their solutions, however, often completely throw away all notions of strict metamodelling. And while we agree that strict metamodelling can be overly restrictive, it certainly has its advantages in protecting ordinary users and simplifying algorithms. So in contrast to tools like XMF-Mosaic (Clark et al. 2014), who completely flatten the modelling hierarchy, we still enforce strict metamodelling, though users can switch to the "unrestricted mode" by taking on a different linguistic type model. Since the unrestricted mode is at a much lower level of abstraction than the usual linguistic metamodels, users will now have more powerful tools at their disposal, and are able to circumvent strict metamodelling in a controlled way.

Third, many tools rely explicitly on the implementation level. For example, MMINT (Di Sandro et al. 2015), MetaDepth (de Lara and Guerra 2010a), DISTIL (Manzanares et al. 2015), AToM^3 (de Lara and Vangheluwe 2002), and AToMPM (Syriani et al. 2013) all explicitly allow users to inject code, for example as parts of models, or to extend the capabilities of the tool. This code is not explicitly modelled, and is simply injected in the actual application code that is being executed. There is thus no checking as to what is happening and if the inserted application code is actually valid code, since it is only treated as mere text by the tool. This code is subsequently only checked by the compiler or parser of the language that is being used, further delaying user feedback. And since this code is dependent on both the application interface (API), and the implementation language, and the internal data structures, the code is not portable at all. Furthermore, it does away with the notion of "model everything explicitly", as it introduces unmodeled aspects in the models and even in the tool. The importance of the physical dimension was previously highlighted (Kurtev et al. 2002, 2006), where the physical storage was mentioned as a technological space. Different ways of representing this data were presented, though each of these can, with our approach, be abstracted away as an implementation detail.

Similarly, megamodel management (Salay et al. 2015) is often implemented purely at the implementation level instead of explicitly modelled. And while there is some work on making generic model management possible (Vignaga et al. 2009; Rose et al. 2011), these approaches often remain specific to the problem under study.

7. Conclusion

We observed that current (meta-)modelling tools have a very strong reliance on their implementation level through the use of code fragments in the implementation language (*e.g.*, Java code as an action in a Statechart transition), explicit reliance on the internal data structures (*e.g.*, XMI), by hardcoding tool semantics (*e.g.*, model management operators in Java), hardcoding semantics of model attributes (*e.g.*, a fixed potency attribute that is always there and further restricts instances), or a combination thereof.

This results in models specific to the specific implementation language, the specific tool semantics, and even implementation details such as data structures. While this is is an easy solution with high performance, these disadvantages are significant.

It is, however, not trivial to explicitly model the complete tool within the tool itself, due to many constraints imposed on metamodelling, in particular strict metamodelling. And while some previous approaches have completely done away with strict metamodelling, we believe that there is still huge value in strict metamodelling, and that it should be kept.

To tackle this problem, we shifted relevant parts from the physical dimension over to the linguistic dimension. The internal data structure was made explicit, such that it can be altered in an explicitly modelled way. To further break the link between these tool services and the internal data structure, which should be an implementation detail, we allow for the actual linguistic and physical metamodel to differ, as long as there is a mapping between them. Operations will be performed on the linguistic metamodel, which the used data structure should translate to its own operations.

These changes allow on one hand to model all algorithms explicitly over a metamodel that is known to be capable of representing each possible model, and is known never to change, even when the internal data structure changes. On the other hand, it also allows us to cope with strict metamodelling by switching to a different linguistic metamodel representation. In this low-level linguistic representation, all data is shown as a single graph, as it was also the case in the physical dimension, such that every possible link is known to be within the same level by definition: every element is at the same level anyhow.

We envision multiple directions for future work. First, we intend to completely bootstrap our used tool, now that the conceptual limitations have been removed. Currently,

the tool still relies on a Python implementation for the action language interpreter, which should now be shifted to an interpreter in action language itself. Subsequently, a code generator should be defined to export action language to any other implementation language. This is very similar to the way Squeak (Ingalls et al. 1997), an efficient and self-describing Smalltalk (Goldberg and Robson 1983) interpreter, was defined. We also wish to further investigate the relation of our tool to Smalltalk. Second, we only considered physical and linguistic conformance in this paper, whereas we left ontological conformance untouched. Ontological conformance might, thanks to multiple conformance relations and the break from the implementation level, also be integrated in this vision. Third, decoupling the internal data structures from the tool itself makes it possible to dynamically change data structure depending on access patterns, similar to the use of activity in simulation tools (Van Tendeloo and Vangheluwe 2014). Fourth, after bootstrapping our tool, implementations in different implementation languages should be easy to create, simply by defining another code generator. We intend to generate our tool for several different platforms, to clearly show that there is no dependence whatsoever on the platform. Finally, we believe that the $conformance_{\perp}$ relation is an enabler for megamodelling (Bézivin et al. 2005), and want to investigate this further.

Acknowledgments

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

OMG OCL. `http://www.omg.org/spec/OCL/`, 2014.

C. Atkinson and T. Kühne. Strict profiles: Why and how. In *Proceedings of UML*, pages 309–322, 2001.

C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

C. Atkinson and T. Kühne. Concepts for comparing modeling tool architectures. In *Proceedings of MoDELS*, pages 398–413, 2005.

C. Atkinson, B. Kennel, and B. Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Proceedings of Semantic Web Enabled Software Engineering*, pages 1–15, 2011.

C. Atkinson, R. Gerbig, and T. Kühne. Opportunities and challenges for deep constraint languages. In *Proceedings of the 15th International Workshop on OCL and Textual Modeling*, 2015.

B. Barroca, T. Kühne, and H. Vangheluwe. Integrating language and ontology engineering. In *Proceedings of MPM 2014 Multi-Paradigm Modelling Workshop*, pages 77–86, 2014.

J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, 2005.

T. Clark, C. Gonzalez-Perez, and B. Henderson-Sellers. A foundation for multi-level modelling. In *Proceedings of MULTI 2014 Multi-Level Modelling Workshop*, pages 43–52, 2014.

J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *Proceedings of TOOLS*, pages 1–20, 2010a.

J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates, and mixin layers. In *Proceedings of MoDELS*, pages 16–30, 2010b.

J. de Lara and H. Vangheluwe. AToM^3: A tool for multi-formalism and meta-modelling. In *Fundamental Approaches to Software Engineering*, pages 174–188, 2002.

J. de Lara, E. Guerra, and J. Sánchez Cuadrado. A-posteriori typing for model-driven engineering. In *Proceedings of MoDELS*, 2015.

A. Di Sandro, R. Salay, M. Famelis, S. Kokaly, and M. Chechik. MMINT: A graphical tool for interactive model management. In *Proceedings of MoDELS*, 2015.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

B. Henderson-Sellers, T. Clark, and C. Gonzalez-Perez. On the search for a level-agnostic modelling language. In *Proceedings of Advanced Information Systems Engineering*, pages 240–255, 2013.

D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of OOPSLA*, pages 318–326, 1997.

I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: an initial appraisal, 2002.

I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *Proceedings of OOPSLA*, pages 602–616, 2006.

C. C. Manzanares, J. S. Cuadrado, and J. de Lara. Building MDE cloud services with DISTIL. In *Proceedings of CloudMDE*, pages 1–6, 2015.

L. Rose, E. Guerra, J. de Lara, A. Etien, D. Kolovos, and R. Paige. Genericity for model management operations. *Software and Systems Modeling*, 12(1):201–219, 2011.

R. Salay, S. Kokaly, A. Di Sandro, and M. Chechik. Enriching megamodel management with collection-based operators. In *Proceedings of MoDELS*, 2015.

E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations*, 2013.

S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne. Multi-level modelling in the modelverse. In *Proceedings of MULTI 2014 Multi-Level Modelling Workshop*, pages 83–92, 2014.

Y. Van Tendeloo and H. Vangheluwe. Activity in PythonPDEVS. In *Proceedings of ACTIMS*, pages 2:1–2:10, 2014.

Y. Van Tendeloo and H. Vangheluwe. Explicit type/instance relations. Technical report, University of Antwerp, 2016.

K. Vanherpen, J. Denil, I. Dávid, P. De Meulenaere, P. J. Mosterman, M. Törngren, A. Qamar, and H. Vangheluwe. Ontological

reasoning for consistency in the design of cyber-physical systems. In *Proceedings of Cyber Physical Production Systems*, 2016. (under review).

A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière. Typing in model management. In *Proceedings of International Conference on Model Transformation*, 2009.

T. Vogel, A. Seibel, and H. Giese. Towards megamodels at runtime. In *Proceedings of Models@run.time workshop*, 2010.