Explicit Type/Instance Relations

Yentl Van Tendeloo
University of Antwerp, Belgium

Yentl.VanTendeloo@uantwerpen.be

Hans Vangheluwe
University of Antwerp, Belgium
McGill University, Canada
hv@cs.mcgill.ca

***Abstract*—The basic building block for constructing a modelling tool architecture, is the relationship between a type and its instances. It is this relation which gives rise to the hierarchy that forms the foundation of the four-layer-architecture and to multi-level modelling. Only through the type/instance relation, a distinction is made between a model and its type model. This relation consists of two equally important components: instantiation and conformance. As both form the foundation of a (meta-)modelling tool, they are often hardcoded, both for conceptual and performance reasons. While this seems logical, it constrains users to the problems envisioned by the tool developers. It becomes necessary to alter models that are not a perfect fit for the provided framework, increasing accidental complexity. Incidentally, minimizing accidental complexity is one of the core goals of Model Driven Engineering. In this report, we consider the limitations imposed by a hardcoded conformance relation. We also present our approach of explicitly modelling the conformance relation: users can chose which conformance to use, and gain insight in the semantics of the tool. We discuss the advantages of this approach, and how this was implemented in our tool: the Modelverse. An example is given where different notions of conformance are used for both structural and nominal subtyping.**

I. Introduction

The basic building block for constructing modelling tool architectures is the relationship between a type and its instances [1], [2]. It is only through this relation, that a distinction can be made between a model and its associated type model. Effectively, a metamodelling hierarchy consists of a set of models, between which the type/instance relation holds. The type/instance relation is bidirectional: going from the type model to the model, there is instantiation, and going from the model to the type model, there is conformance. There is no doubt that this is a special kind of relation, which is of critical importance to the tool. For example, strict metamodelling [3] does not allow relations between a model and its type model, *with the exception of the typing relation*.

But due to the special attention given to these relations, it is often shifted to the implementation level. This is done for a multitude of reasons, the most obvious ones being conceptual clarity (the whole tool depends on it) and performance (it is a frequent operation). Most (meta-)modelling tools store the typing somewhere internally in a datastructure, without exposing it to the user. Similarly, instantiation and conformance checking semantics are hidden from the user. While this is a working solution, such important aspects are hidden inside of the (hardcoded) implementation, with only (possibly non-existent) documentation to guide users. For example, what is the semantics of the typing relation? When can one consider a model to be typed by another type model? And how is it possible, in a uniform way, to change or read out the type of elements?

It has been shown that many alternative conceptual frameworks and implementations have been proposed [1], each with their distinct advantages and disadvantages. Since current tools fix their used framework and type/instance relationship, they become unusable in different domains. This kind of inflexibility was previously identified as one of the major shortcomings of current tools [4].

In this paper, we touch upon these, and other, problems, which we believe are raised by the implicit, hardcoded type-/instance relation. We propose to explicitly model the type/instance relation, by pulling it out of the coded implementation, and making it user-accessible. This includes the type mapping, but also the instantiation and conformance checking semantics. To prevent the extension of the tool interface, and further increase the applicability of the features of our tool, both the type mapping and the semantics will be models in their own right. As they are models, they can become subject to all features offered for ordinary models, such as model versioning, model transformations, and model management operations in general. We discuss both advantages and disadvantages of our approach, followed by an example application.

The remainder of this paper is structured as follows. Section II elaborates on the type/instance relation, and the limitations of the common approaches in use nowadays. Section III presents our solution to these problems: the explicit modelling of both instantiation and conformance checking semantics, combined with an explicit type mapping model. An example application is given in Section IV, where different kinds of subtyping are implemented in our tool. Related work is presented in Section V. Section VI concludes the paper and presents future work.

II. Type/Instance Relation

A language consists of four main components: an abstract syntax, concrete syntax, semantic domain, and a semantic mapping. Of chief interest to this paper, is the abstract syntax, which defines the set of allowed constructs in the language. Generally, this set is described through the use of a type model, defining the allowable types to use, their interconnection, multiplicities, and cardinalities. The type model, however, is limited to structural constraints (*e.g.*, in Petrinets a place can have outgoing links only to transitions), and does not consider the static semantics (*e.g.*, in Petrinets a place cannot have a negative number of tokens). A type model is therefore often augmented with constraints, expressed using a constraint language such as the Object Constraint Language (OCL). This

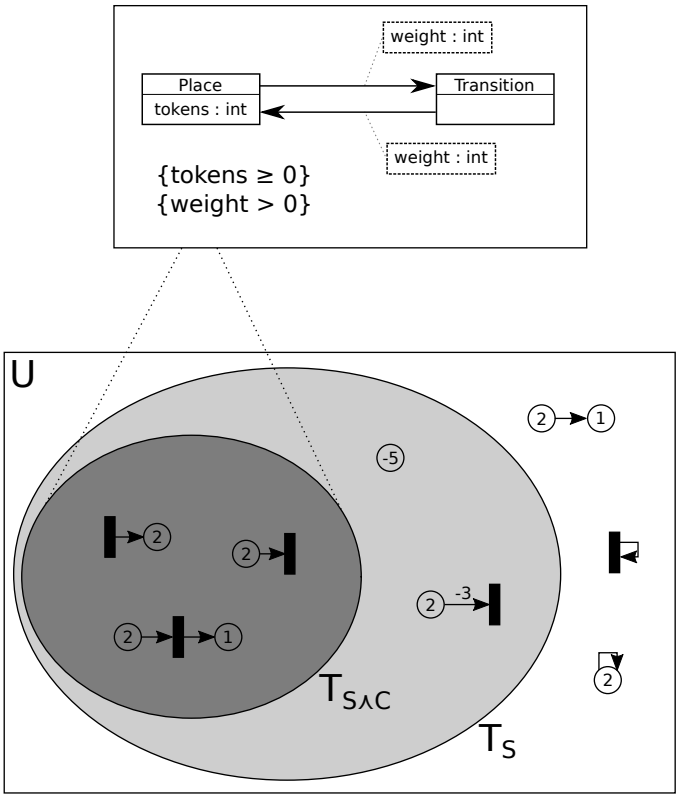

Figure 1: Instantiation of the Petrinets metamodel. The specification on top is *intensional*, whereas the specification at the bottom is *extensional*. U represents all possible instances, including those violating the structure; T_S represents all instances that conform structurally, but not necessarily to all the constraints; $T_{S \wedge C}$ represents all instances that conform both structurally and fullfills all constraints.

is visualized in Figure 1. The totality of this type model can be translated to a single set of constraints [5].

Description of all instances through the definition of constraints is often called an *intensional* description [6]. The other option is *extensional* description, where each instance is explicitly enumerated. For example, the intensional description $\{i \in \mathbb{N} : 0 < i < 5\}$ is equivalent to the extensional description $\{1, 2, 3, 4\}$.

A. Bidirectionality

The type/instance relation consists of two relations, in opposite directions: instantiation (creating a new model from an existing type model), and conformance (checking an existing model with an existing type model). One is the inverse of the other, as each instantiated model should conform to the type model it was instantiated from: $\forall i \in [\![TM_{type}]\!]_{inst} : conf_{type}(i) = true$

Going from the type model to the model, is called instantiation. Through instantiation, an instance is created that conforms (structurally) to the provided type model. A general instantiation method cannot be created, as it strongly depends on what the conformance relation checks, and how it expects the model to be physically represented. For example, if the conformance relation supports subtyping, instantiation should instantiate the own attributes, but also all inherited attributes.

From the model to the type model, we have the conformance relation, often called "verify" in tools (*e.g.*, metaDepth [7] and AToMPM [8]). This function takes a model, a type model, and a type mapping between the two (some parts might be defined implicitly), and maps it to a boolean. Formally: $conf_T : T \times type_T \times (2^T \rightarrow 2^{type_T}) \rightarrow Boolean$. Optionally, insight can be provided in why a model does not conform (*e.g.*, too many instances of a class). A model is said to conform to its type model if each element of the model is an instance of (conforms to) an element of the type model. What it means for a single element to conform to another, is much more vague. For example, if the conformance relation supports subtyping, an element might conform to types other than its own type, if there is an inheritance relation between them.

If one wants to explicitly model the type/instance relation, as is the goal in this paper, one needs to explicitly model both the instantiation and the conformance checking functions.

B. Metamodelling Hierarchy

The metamodelling hierarchy, as popularized by the OMG in the four-layered architecture, makes explicit use of the relation between types and their instances. This architecture consists of four layers, as shown in Figure 2: the metametamodel ($M3$), the metamodel ($M2$), the model ($M1$), and the real world ($M0$). Each of these models is said to conform to the model at the layer above, meaning that the lower level is an instance of the higher level. As a result, $M1$ conforms to $M2$, which conforms to $M3$. At the top of the hierarchy, $M3$ is made to conform to itself, which is called meta-circularity.

This four-layered architecture is used by most (meta-)modelling tools, with only two levels accessible to users: the $M2$ and $M1$ level. $M3$ is fixed, and has a close relation to the internals of the tool, as well as the physical representation of models. $M0$ cannot be modelled in the tool, as this represents the real world instance. This leaves only $M2$ and $M1$ for modification. Users can then use $M2$ to define their own custom language, specific to the domain they are interested in. $M1$ is used to model the actual model that is to be manipulated.

Having only two levels at your disposition can be limiting for several applications [9]. Therefore, multi-level modelling has been introduced, where the number of user-accessible layers is unrestricted. This raises the question on how to restrict elements several layers deep, which can be done through the use of deep characterization (*e.g.*, through potency [6], [10]). These techniques have an influence on instantiation and conformance semantics, and therefore require specialized tools. For example, potency will prevent the instantiation of elements whose potency value has reached 0. The exception being potency *, which indicates unrestricted instantiation until specified [11].

C. Hardcoded Type/Instance relations

From the previous examples, it becomes clear that instantiation and conformance are more complex than graph (or

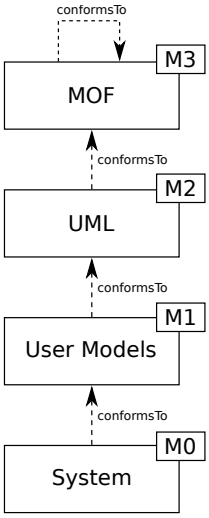

Figure 2: Four-layered architecture, exemplified using Petri nets.

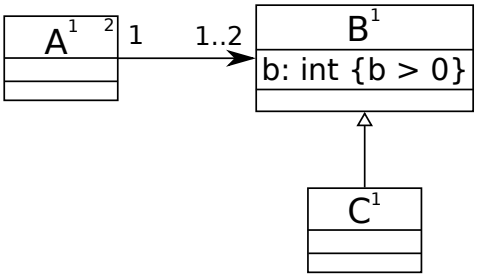

Figure 3: Constructs that further restrict the set of instances.

model) homomorphism. Apart from finding whether the model structurally conforms to the provided type model, additional constraints are imposed on the representation. Several of the instances of the graph even gain special semantics, deviating from their originally structural role. For example, a *cardinality* attribute will not merely be structural, but will further constrain the set of allowed instances. Figure 3 shows some more examples of additional constraints.

Their semantics is as follows:

- *Inheritance*. A special kind of link between two elements. It cannot be further instantiated. The source of the link becomes a subtype of the target during the conformance check. In the example, this means that any instance of C will also be considered as an instance of B, but not the other way around. Additionally, when instantiation an instance of C, all attributes of B also need to be added.
- *Potency*. A special attribute which indicates how many levels deep this element can still be instantiated. When instantiating an element, the potency value is copied and decremented by one. When the value reaches zero, no further instances can be made. In the example, this means that both A, B, and C can only be instantiated once more. This places restrictions on both instantiation (*i.e.*, refuse to instantiate) and conformance checks (*i.e.*, always find them to be non-conforming).
- *Cardinalities*. A special attribute of an association which limits the number of instances of this association for a single element at the other side of the association. Both a lower and upper bound are possible. In the example, this means that each instance of A has either 1 or 2 connected instances of B through this association. For each B, there is exactly one connected instance of A. Instantiating additional links, when the upper limit is already reached, should be disallowed. Conversely, a conformance check should flag a model as non-conforming if the constraint is violated, even though it is structurally fine.
- *Multiplicities*. A special attribute that indicates how many instances of this element can be present at the level immediately below. Both a lower and upper bound are possible. In the example, this means that there will be exactly 2 instances of A. Similarly to cardinalities, both the instantiation and conformance relation should be aware of these restrictions.
- *Constraints*. Previous constructs would limit the structure, whereas this part restricts the instances based on the value of attributes, often refered to as *static semantics*. Arbitrary executable models can be coupled to the type model, which are evaluated when determining whether the element conforms or not. In the example, this means that the value of the attribute b of B will always be greater than zero. The conformance check should, apart from checking the previous constraints, execute this piece of code to determine whether or not the model conforms. Instantiation does not need to be aware of this, as there is no way to statically know which operations are allowed to satisfy this function. Apart from the local constraints, as specified in the example, global constraints exist as well. These constrain the instances depending on a combination of multiple elements of the instance.

When these constructs are only present structurally, as is the case in most modelling tools, their semantics is non-obvious. It might even be non-obvious whether or not the attributes have any semantics (within the modelling tool) at all: why would an attribute with a specific name suddenly become part of the restrictions placed on instances? Somewhere, semantics needs to be given to these constructs: a component of the tool needs to find the attribute, read it out, determine whether or not the model satisfies this requirement, and provide user feedback.

While we acknowledge that these powerful constructs aid in creating a tightly constrained set of possible instances, their inclusion often creates severe problems in the modelling hierarchy. Because each of these carries its own semantics, informally described above, there needs to be a mechanism to enforce the semantics. As described above, this semantics is part of the type/instance relation, which is, in most current tools, implicit and hardcoded.

More concretely, this results in the following problems:

1) *Semantics*. The exact semantics of these constructs is often unclear [3], and only found out by reading

documentation or through experimentation. While for some constructs the semantics doesn't vary much between tools, other constructs vary significantly. And even if the semantics is clearly communicated between both parties, it remains a problem as to how these semantics are applied by the instantiation and conformance checking operations. For example multiplicities: what if a lower bound is not reached? Does it become impossible to delete elements, which would cause this lower bound to be violated? Or would a deletion be allowed, but subsequent conformance checks do fail in case it is still violated at that point in time? And is it possible to save a model which violates these constraints? Similarly, is it possible to create additional elements if these would violate the constraints? Or is it only possible within some kind of a transaction?

This is even the case in object-oriented programming languages, where the semantics of subtyping varies. For example, C++ offers multiple inheritance, whereas Java only offers single inheritance. On the completely opposite side of the spectrum, Haskell uses structural subtyping instead of nominal subtyping [12]. While each of these has its advantages and disadvantages, it should be clear to users which semantics are used.

2) *Static*. Semantics, even if formally described in the documentation, still remains static. While this is not a significant problem in general, as a general concensus exists for these attributes, sometimes a slightly different semantics is desired. For example, users might want to make to temporarily violate a multiplicity constraint if the restriction is too strict.

   Similarly, some users might prefer, or even require, different semantics than those implemented. For example, Java limits inheritance to single inheritance. Users that require multiple inheritance will have to resort to tricks to implement their models which naturally lend themselves to multiple inheritance. Should the semantics be modifiable at run-time, users can alter the behaviour to their liking, or just switch implementation. Users who prefer to be constrained to single inheritance, can then use the single inheritance semantics, whereas others can decide to opt for multiple inheritance.
3) *Special constructs at the implementation level*. Because some constructs gain a special semantics, there needs to be a way of identifying these constructs. For some this is easy (*e.g.*, read out an attribute with a pre-defined name, such as *potency*), but for others, this becomes more difficult. In particular, the inheritance relation is a special case: it is a link, and one would expect it to be implemented as such. Many frameworks [13], [14], [8], however, rely on this (or similar relations) to be a special kind of link, unrelated to a normal association. And while their underlying model storage hugely mimics existing structures, such as graphs, exceptions need to be made throughout to cope with these constructs. Furthermore, this additional type causes further problems in the checking of conformance: how is it typed? Resolving these elements should not be done through hardcoded types at the lowest level. This prevents the reuse of existing libraries, as a wrapper needs to be written to cope with the special types.
4) *Special constructs at the metamodel level*. Even if special constructs at the implementation level are avoided, special constructs at the metamodel level are sometimes still used. This hardcodes the identity of some parts of the metamodel in the instantiation and conformance checking functions. The metamodel will therefore simply be a normal metamodel, though some associations will gain special importance which are not apparant from the metamodel alone. It is only the type/instance relation which adds this additional semantics to the link.

   Apart from the confusion this might cause to users, it prevents users from using a different metamodel, and even prevents multi-level modelling completely. This was one of the problems that prevents AToMPM [8] from having multi-level metamodelling, or just more than two metametamodels: the inheritance semantics is hardcoded in the core, and only applies to the provided metametamodels.

   Similarly, models cannot simply have attributes if their metamodel does not allow for it. While this is not a problem in the traditional four-level architecture, multi-level modelling quickly runs into this problem: users can only specify a potency if their metamodel explicitly calls for it. Instead of modifying the metamodel for this, it is possible to encode these *special* constructs as "explicitly allowed" in the conformance relation, as was done in the previous (unrelated) version of our tool [15]. The instantiation algorithm should also be aware of this, as it should, for whatever model, always add in these attributes by default, and furthermore make them mandatory, such that they cannot be removed.
5) *Inflexible type mapping*. Type mappings store the types of elements. While they have previously been identified theoretically, most current approaches hide away this important piece of information in the implementation. Apart from reading out the type, and possibly altering it through some programming interface, no modifications are possible as they reside in the internal data structures of the tool.

   By making these type mappings explicit, as an ordinary model, it can be modified as any other model, and in particular through the use of model transformations. This is one of the limitations of model transformations: the right-hand side cannot create an instances of a metamodel that is only known at runtime. This problem is currently solved by using model transformation templates [16], which is still more constraining than our approach, as it doesn't allow for retyping operations.
6) *Single type*. Finally, as the conformance function and typing information is hardcoded, only a single such relation is possible for a given element. Sometimes, however, an element can be typed by multiple, possibly unrelated, elements. An example has already been given in [17], where a model is created with a single (constructive) type, but additional types can be found during execution. This could however also be

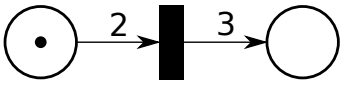

Figure 4: Petri net model (concrete syntax for readability) that will be encoded.

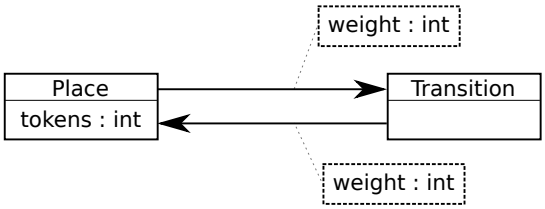

Figure 5: Metamodel (concrete syntax for readability) of Figure 4.

related to the use of multiple very similar metamodels. For example, consider a petri nets metamodel, and a seperate, but identical, petri nets metamodel with inhibitor arcs. Every petri net instance without inhibitor arc, also conforms to the petri nets metamodel with inhibitor arcs. Similarly, every petri net without inhibitor arcs, even if it was constructed as an instance of the metamodel with inhibitor arcs, will conform to the original metamodel. Even though these are unrelated, a model can easily be said to be typed by both of them, depending on the situation in which it is used. This can have further repercussions in model evolution, where models frequently need to be retyped to slightly different metamodels.

III. Explicit Type/Instance Relations

We now present our solution to the previously identified problems. As all problems were caused by the hardcoded algorithms, our approach explicitly models the type mapping, the instantiation algorithm, and the conformance check algorithm. For this, we introduce (1) a semantics-free representation of models (and subsequently type models), (2) an explicit type mapping model, (3) an explicit instantiation algorithm and corresponding (4) conformance algorithm in an executable modelling language. We also show how this approach naturally allows for multiple type models. Finally, we present the advantages and disadvantages of this approach, when compared to hardcoded type/instance relations.

We use a simple petri net model and corresponding type model, shown in Figure 4 and Figure 5, respectively, to illustrate our approach.

A. Models

Since all semantics need to be shifted to the type/instance relation, the model representation becomes essentialy semantics-free. As there are no longer any attributes with special purpose (and should thus always be there), nor are there special kinds of elements (such as inheritance links), the representation boils down to a simple graph. Because now the complete structure of the model can be described only through nodes and edges, existing graph databases can be reused. Apart from the applicability of previously defined algorithms, this lowers the burden of users trying to understand how and what data is stored.

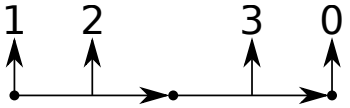

Figure 6: Petri net model representation in the Modelverse.

As there is also no longer any distinction between the semantics of models (cannot be instantiated) and type models (can be instantiated), both are reduced to the same representation. This further unifies the core implementation of a (meta-)modelling tool, and avoids the problems previously identified by storing a model twice [1].

For our example, Figure 6 shows how Figure 4 is represented in the Modelverse as a graph. The graph contains only structural information on what the instance will look like, and doesn't contain any attributes that have semantical meaning in the context of the type/instance relation. Note also that the instance model does not contain names on the links, but only their values. It is the type model, and corresponding typing relation, which gives the name to the attributes, making them identifiable. This was done for several reasons, but most importantly, this allowed the graph structure to be more restrictive, as it explicitly stores the names of the allowable attributes. Additionally, this is very similar to how most general purpose object-oriented programming languages work. Furthermore, it makes the stored data more independent of the names used in the type model, such that, for example, different names for the same attribute can be used interchangeably (*e.g.*, due to translation or different terminology).

B. Type Mapping

To explicitly represent the typing relation, they need to be represented as a model. They are not to be included within the model itself (*e.g.*, as some kind of association) for three reasons:

1) Making the distinction between normal links (instances of associations) and typing links becomes hard. This bears significant similarities to the problem we were trying to avoid with the inheritance links being of a special kind. As such, creating direct type links directly needs to be avoided as they would otherwise qualify as normal links. Unless, of course, our instantiation and conformance algorithms can cope with this.
2) Only a single type is possible if it is linked directly to the elements. It would be possible to create multiple type links starting from a single element, for example have two outgoing type links for a single element. These two different type links, however, would be ambiguous, as it is unknown which mapping they belong to. Type links are frequently interrelated: if there are two elements, with both having two possible types, a total of four different combinations become possible. As this depends on the situation, it is

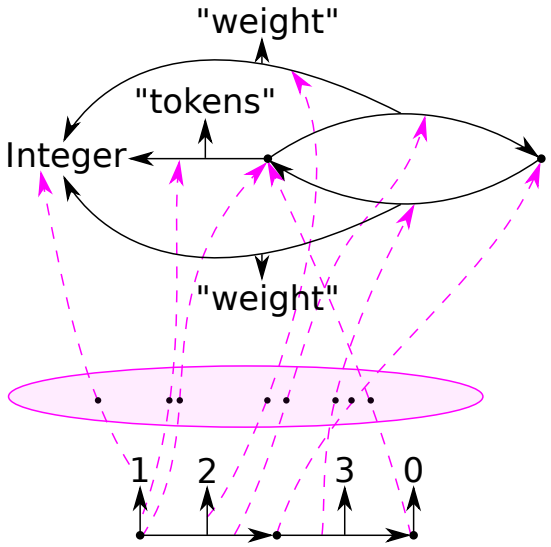

Figure 7: Petri net typing relation with dashed lines (excerpt).

impossible to create general assumptions on this, and should therefore be avoided.

3) Type links should ideally be stored in a seperate model, such that they can have their own constrained type model. If these links were part of the original model (as outgoing links), they would need to be part of the type model of the model. With type mappings as seperate models, they can have a simple type model, which largely resembles a kind of dictionary.

To circumvent these problems, we have opted to create seperate type mapping models. These models are rather similar to a dictionary, where the keys are the model, and the values are the type of the model. And while they resemble a dictionary, they are explicitly modelled, and thus user-accessible. Users can thus open this type mapping just like any other model, and modify it if desired. In particular, model transformations can now also query the type of elements, or create elements of specific types, by manually modifying this dictionary.

Figure 7 presents an excerpt of a possible typing relation between the petri net model and type model. Most importantly, the typing relation can be accessed as a single node (root node of all typing links), making it easy to use a different one.

C. Instantiation Algorithm

When an instance is made of a type model, several things need to happen. Most importantly, the instance itself needs to be created. Furthermore, however, the instance needs to be registered in the type mapping, effectively updating two seperate models. Also, the operation needs to be checked for validity: is it even possible to perform this operation?

Updating the type mapping is intimately related with the representation of the type mapping, which we have, up to now, proposed as inherent to our approach. Any possible relation, however, is possible with our approach. As such, a type mapping in itself is not mandatory, but it is only an artifact of our example type/instance relation. Type/instance relations without a type mapping, or with a very different one, are possible. Therefore, information on the type mapping, such as its representation and encoding, is necessarily included in the instantiation algorithm.

Similarly, information on the semantics of additional constraints is necessary. The instantiation algorithm needs to be aware of the things that need to be checked when a new instance is created, such as potency, cardinality, multiplicity, and so on. Attributes can also be read out from the type model (and possibly supertypes of the found type), and presented to the user. Users would then, instead of manually specifying the name of the elements to create, be provided with a list they need to fill in, similar to AToMPM [8]. All these operations need to be explicitly defined in the instantiation algorithm, where they are available for users to look up or modify. Depending on the front-end the user uses, different instantiation algorithms might be ideal. Indeed, a textual front-end that runs in batch should not prompt the user, whereas an interactive visual modelling environment should prompt users if information is missing.

For our example, this means that the instantiation algorithm will only ask users which element they want to instantiate (*e.g.*, *Place*), and give it (optionally) a name (*e.g.*, *p1*) for later reference in the model. Afterwards, users can specify attributes to instantiate (*e.g.*, *tokens*), for which the algorithm will automatically resolve the types from the type model and subsequently check for conformance to the required type.

D. Conformance Algorithm

Finally, an algorithm needs to be devised which takes a previously defined model, type model, and mapping between them, and determines whether or not the model conforms. This algorithm will, for each element in the model, check whether the type mapping points to an element in the specified type model. Additional constraints, such as potency and cardinalities, also need to be checked. For each edge, the source and target are checked: the source (target) of the model needs to be an instance of the source (target) of the edge in the type model. To determine whether an element is an instance of another element, we consult the type mapping. In addition to “direct types”, it is possible for an element to be a subtype. Inheritance links are therefore followed during the conformance check, finding out the relation between the found type, and the expected type. Recall that there was no longer any way of identifying the inheritance link at the physical level, as it was just another association. For this reason, this specific conformance algorithm takes an additional parameter: the *inheritance* association. This is the *type* of each inheritance link, of which the instances are the actual inheritance links. The conformance algorithm therefore only takes a single inheritance association as parameter, and can automatically find all of its instances, the actual inheritance links. Inheritance semantics is provided by the conformance algorithm, which knows that following inheritance links is allowed when finding instances.

The conformance algorithm also searches for constraints to execute, multiplicities and cardinalities to check, and potencies to update. All semantics is now explicitly modelled in the conformance algorithm, resulting in several degrees of freedom. For example, it becomes possible for users to encode

any of these restrictions wherever they seem best suited. We avoid the need for every model to have a mandatory attribute, like *potency*, as this is up to the conformance algorithm to decide. Other alternatives are equally valid, as long as they are explicitly modelled in this algorithm. Our approach offers much more flexibility to the users, and allows for models better suited for the problems they are trying to solve.

For our example, this means that the conformance algorithm will read out all elements of the model and check whether they are typed correctly: is the *tokens* attribute indeed an *integer*, is there no edge going directly between two *place*s, etc.

E. Multiple Metamodels

With all pieces into place, we can now discuss the possibility for multiple type models. As each aspect of the type/instance relation is explicitly modelled, and thus accessible by the user, models can conform in different ways. For example, users might provide a different type mapping, a different type model, or a different conformance algorithm altogether. We will only provide a simple example, which was already hinted at, where a single Petri net model conforms to two distinct type models: normal place/transition nets and place/transition nets with inhibitor arcs. The conformance algorithm, when passed with two different type mappings and corresponding type model, will state that a petri net without inhibitor arcs, conforms to both type models. This is shown in Figure 8. A petri net containing at least one inhibitor arc will only conform to the type model with inhibitor arcs. Further differences between the type models are possible (even structurally, by using a different conformance check), though these are not shown here to prevent confusion.

One of the remaining problems is one of consistency: both type mappings are updated independently, and should also be maintained seperately. As a result, if users add an additional place to the model, they would have to update both type mappings. If a type mapping was not, or incorrectly, updated, subsequent conformance checks will fail until the problem is resolved.

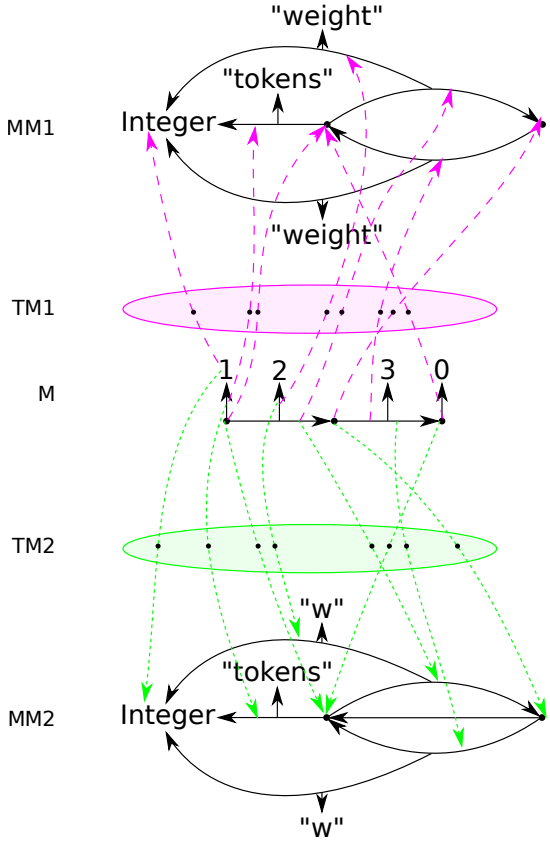

Figure 8: A single petri net model conforming to two different type models simultaneously. The top type model is without inhibitor arc, whereas the bottom one has an inhibitor arc. Names of attributes also vary slightly.

F. Advantages and Disadvantages

As with everything, our approach implies some disadvantages, mainly related to usability:

1) *Explicit management*. While the explicit modelling of the type/instance relation has its advantages, users might be bothered with the additional complexity. We believe, however, that this complexity can be hidden (though accessible) from users who do not require it (novice users), and only accessible by advanced users. Whereas other tools simply offer a built-in instantiation and conformance function, users now have full control over this, since it is just another function. This does not necessarily need to be a disadvantage: the conformance function is considered as any other function, and can also be used as such. Use of specific APIs is thus avoided, and users have less need for documentation on what exactly is this function and how it is implemented, as they are already familiar with how to use it (just like any other function), and the semantics can easily be seen by opening the relevant model (instead of wading through the source code of the tool).
2) *Managing multiple concepts of type/instance relations*. The concept of allowing for multiple types of type/instance relations is an attractive one, but can also stand in the way of users. We must acknowledge that most users will probably never need to manage the use of different kinds of type/instance relations. This is a trade-off: do we limit the functionality of our tool, such that it is easy to use for all users, or do we open all aspects of the tool, potentially confusing many users? Again, we believe that an adequate interface will help users in managing this complexity.
3) *Tool itself is no longer a complete metamodelling tool*. What generally identifies a tool as a metamodelling tool, is its support for instantiation and conformance, and optionally support for model management operations. By removing all these aspects from the core of the tool, but shifting them a level higher, the tool essentially no longer has support for modelling. Instead, tools become simple model interpreters, which will have to interpret the provided instantiation and conformance algorithms to become capable of modelling. While there is the danger

of becoming too general, this clearly seperates the core of the tool from its additional functionality. Nonetheless, the design and implementation of the tool is still oriented towards metamodelling.

4) *Efficiency*. Up to now, efficiency has not been a significant criteria when evaluating metamodelling tools. While it is true that some are more efficient than others, certainly for extremely large models, most tools cope reasonably well with small to medium-sized models. Interpretation of one of the core functionalities of the tool, however, is detrimental for performance. With naive implementations, tools become too slow to use, even for small models. Currently, we see this as one of the primary limitations of our approach, as it likely necessitates much tweaking of model interpretation performance. Nonetheless, we find this very similar to Smalltalk [18], where most functions are also provided as library functions, written again in Smalltalk. While Smalltalk itself was not efficient, the Squeak [19] environment proved that high speedups are possible, even for this kind of languages.

We believe that these disadvantages can be dealt with by increasing usability in general. A clear syntax greatly aids users in managing this additional complexity. Furthermore, sane defaults should be provided, such that users can hide the complexity if they don't need it. It should be possible to offer users a simple syntax, in which defaults are used, and a more advanced syntax, in which users have full control. From the point of view of efficiency, efficient model interpreters are required, possibly through the use of Just-In-Time compilation (JIT). Previous interpreters have seen significant speedups through the use of JIT compilation, such as Squeak [19] for Smalltalk [18], a language well-known for its philosophy of making every aspect explicit. Compilation of the model might also be possible, such that efficiency becomes comparable to that of hand-crafted code.

Should these disadvantages be overcome, it offers us several advantages. These advantages are related to the previously identified problems of a hardcoded type/instance relation:

1) *Explicitly modelled semantics*. By explicitly modelling the semantics of the tool, it becomes independent of the implementation platform and navigable to users. Users no longer have the need to consult seperate tool documentation to know the semantics: they are explicitly browsable, just like any other model. Furthermore, it becomes managable like any other function, making it susceptible to model transformations or modification. Additionally, users only need to know one language: the modelling language. Previous tools with an explicitly modelled action language, still required users to work with their implementation language to extend the tool (*e.g.*, through plug-ins or extension points).
2) *Dynamic type/instance relation*. The algorithms and type mapping not only become visible to users from within the tool itself, but they can also be modified dynamically. As the function is interpreted, changes are immediately visible to users, stimulating rapid prototyping. We do, however, acknowledge that there should be some restriction to this high degree of freedom, in order to prevent absurd situations.
3) *Use of pre-existing libraries*. By removing the need for special elements at the implementation level, well-known data structures can be used, such as graphs. There is no longer any need to implement specific kinds of graphs (*e.g.*, with special "inheritance" links, or even Typed Attributed Graphs), as every link, even the inheritance and type link, will be an ordinary edge. Instead of through the model database, semantics is given by the interpretation of the algorithms. Many tools and algorithms exist for managing extremely large graphs, forming a research domain on its own. All these tools and algorithms can be used as-is, without any modification or wrappers at all. Very minimal wrappers are still required, to have the tools communicate with each other, though these wrappers don't hold any conversion logic, nor do they alter the semantics of the stored graph.
4) *Multiple possible metametamodels*. As there are no longer any "special" metametamodels, with hard-coded parts in the core of the tool, any model can potentially become a metametamodel, or even meta-circular. Each model that is sufficiently expressible can serve as the new root of a modelling hierarchy. The instantiation and conformance algorithms still need references to the model (*e.g.*, to know about the inheritance relation), but it can be fully customized. So while some changes are still required, these changes stay within the tool, and don't force the metametamodeller to leave the tool even once. Multiple dimensions to conformance exist, as identified in the OCA [20]. In this paper, we limit ourselves to the linguistic dimension, but the need for multiple type models is even stronger in the ontological dimension, where it relates back to properties a given model satisfies [21], [22].
5) *Full support for multi-level modelling*. Taking the previous advantage a step further, any modelling hierarchy becomes possible, as long as the conformance relation is made to cope with it. All attributes that influence conformance, such as potency and cardinality, become explicitly modelled at each level of a multi-level hierarchy. Multiple kinds of instantiation semantics can be implemented, for example using potency [6], or the unified version which also applies to edges [10].
6) *Flexible types*. Type mappings are also explicitly stored as models, making it possible to use them like any other. Possible use cases of this are to query the types of elements, or to modify the types at runtime. Should types be coded somewhere in the core of the tool, this becomes impossible without the use of a dedicated API. Similar to our arguments for the explicit modelling of sementics, reusing existing interfaces is more familiar to users than creating new ones.
7) *Multiple types*. In addition to making it possible to have multiple possible type models, or even metametamodels, a single element can be typed by several different elements of potentially different type models. These type relations are stored in seperate

type mappings, such that the user can decide which typing relation to use for a specific operation.

IV. Example Application: Subtyping Semantics

We now present the power of our approach through an example application. Our example makes use of different notions of subtyping, which is closely related to the type/instance relation. In this application, we mainly focus on determining wheter a model conforms to a type model. A similar discussion is possible for instantiation, though here we assume that the model was already created one way or the other. As current tools hardcode their conformance relations, users have no other option than to use the given type system. We continue by showing how our tool, the *Modelverse* [23], is able to cope with different kinds of type systems. First, we briefly elaborate on the typing problem at hand.

A. Subtyping

Throughout the years, different kinds of subtyping relations have been defined and used. Subtyping directly influences the conformance check: an instance of *Class B* is also an instance of *Class A*, if either *B* is *A*, or if *B* is a subtype of *A*. The definition of *subtype* varies between the different type systems. We briefly present the two major classes of type systems: nominal and structural typing [12].

1) Nominal Typing: The most commonly used kind of subtyping is nominal typing. With nominal typing, subtypes must be explicitly declared, such that there can be no confusion or unexpected behaviour. Most programming languages are implemented like this, with classes that can *inherit* from one another. If *Class IPlace* inherits from *Class Place*, all instances of *IPlace* will conform to both *IPlace* and *Place*. Additionally, the structure of *IPlace* is automatically extended with all concepts of *Place*. So if *Class Place* had an attribute *tokens*, all instances of *IPlace* also have this attribute, in addition to all the attributes that were also provided by *IPlace*.

Nominal typing is most oftenly used in Object-Oriented General-Purpose Languages (GPL), such as C++, Python and Java. Even then, subtle differences can be seen. For example, C++ and Python allow for multiple inheritance (*i.e.*, a single class can inherit from multiple classes), whereas Java restricts users to single inheritance (*i.e.*, a single class can only inherit from one class). Again, several variants exist in semantics: attribute resolution is different between C++ and Python, even though both have multiple inheritance. Even worse, the multiple inheritance semantics in Python has already been changed three times[1] between different (otherwise compatible) versions.

Nominal typing is considered safer, as the typing information can be used both for typechecking, and at run-time. Additionally, checking subtyping relations becomes almost trivial. Flexibility, however, is partly lost: all typing relations should be defined explicitly, which might be impossible when legacy code is used.

[1] https://www.python.org/download/releases/2.3/mro/

2) Structural Typing: At the other end of the spectrum lies structural typing, where subtypes are implicitly detected instead of explicitly defined. A *Class IPlace* is considered to be a subtype of *Class Place* if each feature of *Place* is also present in *IPlace*. Consequently, if *IPlace* and *Place* have exactly the same features, they will be considered equivalent, with all instances of *IPlace* being instances of *Place*, and vice versa.

Structural typing is used by languages such as OCaml, Go, and Haskell. Now again, subtle differences exist, such as whether or not the name of the features need to be identical, in addition to the type. Structural typing is considered more elegant than nominal subtyping, as it lies closer to type systems studied in the literature. While great flexibility is gained, elements might be typed by others "by accident": simply because an element has the same attributes, doesn't mean that they are identically typed.

B. Conformance Relation

Current (meta-)modelling tools, and even programming language compilers, hardcode their type system, and the user has to oblige. While the default type system is often a sane choice for the domain for which the tool was developed, situations will occur where the other choice might have been easier for the user. Users have no choice though, as the semantics of the conformance relation is hardcoded. Even worse, many modelling tools explicitly hardcode special *instanceOf* and *subtypeOf* links inside of their data structure.

With explicitly modelled type/instance relations, typings become modifiable by the user, possibly even automatically using model transformations. In our case, it is further possible to create multiple conformance relations, one for each kind of type system, and use either, depending on the situation. Depending on the conformance relation chosen by the user (ideally, the one closest to the problem domain), the outcome of the conformance check and every related operation will vary.

C. Modelverse

In the Modelverse, users are required to specify which conformance relation they want to use, and where they want this type mapping model to be stored. When a model is instantiated, an instantiation policy also needs to be provided. Later on, when conformance is checked, the previously specified conformance relation is used.

Users still have to implement the conformance checks they wish to use, except if it is provided by default, or by other users. For our example, a subtyping check for nominal typing will follow inheritance links when required, whereas a structural typing check will only look at the features of the class.

Note that, for the case of nominal typing, there are links with a special semantics. In our approach, these links are just normal links, stored like any other link (*i.e.*, as an edge). The conformance check, however, knows the semantics of the links, and uses them appropriately.

In the following example, we define two Petrinet metamodels: one for ordinary Petrinets, and one for Petrinets with inhibitor arcs. A Petrinet without inhibitor arcs can thus

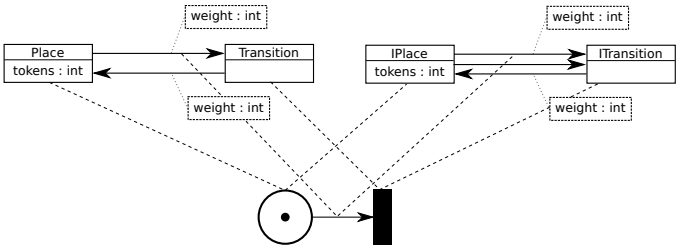

Figure 9: Using structural subtyping, a simple Petrinet instance conforms to both the simple metamodel, and the metamodel containing the inhibitor arc.

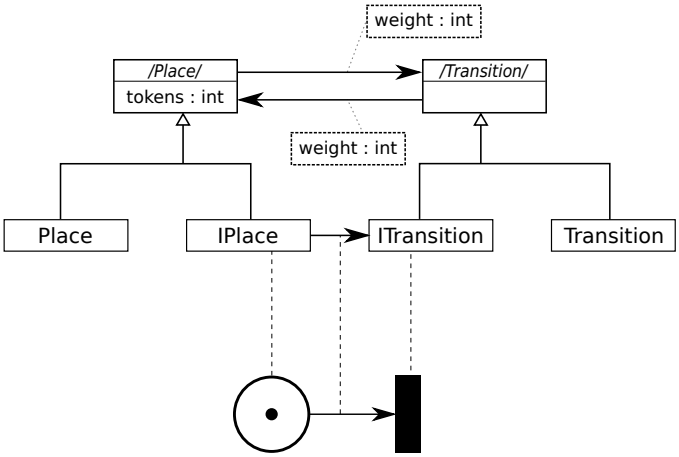

Figure 10: Using nominal subtyping, an explicit inheritance link is necessary to indicate subtyping. Each instance now conforms to exactly one of the concrete classes, but always conforms to the abstract class.

equally be described using either of both formalisms. As a result, such a Petrinet instance should be able to conform to both metamodels at the same time, and operations defined for either of the two metamodels should be applicable to the model. As soon as an inhibitor arc is added, it becomes clear that the model no longer conforms to the ordinary Petrinets metamodel, but instead the one with inhibitor arcs is required.

When we implement this example, we can create both metamodels independently for structural subtyping, as the conformance checking algorithm will automatically detect the similarity. For nominal subtyping, however, an explicit similarity needs to be defined. For this, we defined an abstract Place and Transition, which will serve as the superclass for both the ordinary Petrinets and the Petrinets with inhibitor arcs. In this situation, both metamodels need to explicitly state that they want to be subtypes of the abstract classes.

The results of the conformance check for nominal and structural subtyping are very different, and could alter the complete semantics of the model. Using structural subtyping, operations defined over either the *Place* or *IPlace* would be applicable to the place. This implies that any object that has a *tokens* attribute will be applicable for the function, which is ideal for functions that operate solely on the number of tokens. Using nominal subtyping, all operations that should be applicable to both *Place* and *IPlace* instances need to be defined on the abstract class */Place/*. As such, there is no reuse unless it is explicitly allowed. Furthermore, if *Place* was defined before the *IPlace* extension was thought of, the legacy model (*Place*) will also have to be updated, and all (or most) of its operations ported to the abstract class */Place/*.

While we do not want to argue which one is the best, we offer the user the possibility to use either of these (or others), depending on what the user believes to be the closest match to the problem domain.

In the future, we plan to provide a sane default, similar to the conformance check found in most other tools. This will simplify modelling and prevents users from the associated complexity if this functionality is not needed.

V. Related Work

The problems we solve have already been partially addressed. But while our approach solves several problems simultaneously, by taking a different view on the type/instance relation, other approaches are more conservative, in that they solve only one specific problem by making changes to the current state of the art.

The main motivator for our research was the credo "model everything at the right level of abstraction, using the most appropriate formalism" [24], as popularized in Multi-Paradigm Modelling (MPM). Naturally, this applies to the system being modelled. We, however, also apply this to the tool, which will be used for the modelling, as there would be no reason to consider the tool as a special entity for which this credo does not hold. As a natural step then, the conformance algorithm was made explicitly modelled. It is not only the credo which we take over from MPM, but we also create our tool to be used in an MPM setting. For example, our notion of conformance could possibly be used for multi-view and multi-abstraction modelling. Further research is required in this direction.

While the conformance relation has, to the best of our knowledge, always been hardcoded as foundational aspect of the tool, the type mapping has frequently been made explicit, certainly in theory. For example, de Lara *et al.* [25], [26] have explicitly defined type mappings (*explicit bindings*) for their concepts, which are an alternative form of model typing [27]. VPM [28] made the *instanceOf* links available just like any other element, thus avoiding hardcoded relations. Our typing relation is actually a relation between models, similar to the relations defined for megamodelling (*e.g.*, [29], [30]). Kurtev *et al.* [14] proposes to define a model as a triple, containing the model, the type model, and the type mapping. No mention is made, however, about making this relation depend on the conformance relation. Zschaler [27] already proposed to make explicit all constraints on type models in the model type, rather than hiding them away in the matching relation. We also believe that constraints should not be hidden in the matching relation, but instead of putting it in the model type, we make the matching relation explicit.

Models have also often been represented using a graph representation, for example in VMTS [13], or in the framework proposed by Kurtev *et al.* [14]. Despite their mapping to graphs, they include special edges, such as an inheritance edge. Such edges are not defined in general graph theory, and

should therefore be avoided as it mixes parts of the structure (*i.e.*, the actual graph) with the semantics (*i.e.*, the system we represent with the graph). In our approach, everything is stored as a graph, which purely defines the structure of the models. Semantics is not given by this graph representation: it is given by the executable models, which are themselves models in our tool, represented as graphs.

Closest to our contribution is a-posteriori typing [17], where a single model element has one single (constructive) type, but possibly also a set of additional types which act as views on the model. The constructive type is the one gained through instantiation, and is not modifiable, nor user-definable. Additional types, however, are dynamically determined when used in a specific setting. Whereas in our approach we explicitly model the conformance relation and the type mapping, a-posteriori typing is more of a kind of conditional typing, related to their implementation of concepts. Elements are considered as instances of specific types only if some conditions are met, such as having a specific attribute with a value within a certain range. Its primary use is for reuse of functions, as exemplified in their application on the petri net model. In comparison, our approach offers more flexibility, as an element might be typed by multiple element (similar to a-posteriori typing), but also typed using multiple conformance relations. Since conformance relations are explicitly modelled, there are no limitations as to what the modeller can model.

Similar to our work is also that of the Dynamic Multi-Layer Algebra [2], which also acknowledges the importance of bootstrapping the system and making the instantiation explicit. But whereas the instantiation is explicitly mentioned in the paper and the algebra, there is no way for users to modify or extend it at run-time.

Retyping is offered by other tools, either through model transformations as in AToMPM [8] and MetaDepth [7], or by seperate operations as is the case in MMINT [30]. Instead of retyping, our approach has multiple types simultaneously, thus avoiding frequent type conversions at the cost of managing all type relations simultaneously. Retyping is also possible, without the need for seperate operations, as the typing relation is just another model.

Related to multiple types for a single element, Nivel [31] allowed for multiple classification. In contrast to our approach, only a single kind of conformance relation was defined, which was directly encoded into their formalization and is thus a fundamental part of the tool. Similarly, they hardcoded concepts like subtyping, reducing flexibility further.

Several hardcoded conformance relations were proposed, such as that of Nivel [31] and by Guy *et al.* [32]. Each of these explicitly requires some attributes to be present, such as multiplicities. In general, it is not guaranteed that the type model will even contain these notions, let alone that they have identical semantics. By making the semantics of these elements explicit, users are certain about what the values mean, but additionally they can also opt for simpler type/instance relations, which don't take into account some of these restrictions. Salay and Chechik [33] raised the need for modification (relaxation) of the conformance relation to allow for agility.

In some way, de Lara *et al.* [25] propose a different kind of conformance check, though still hardcoded: using requirements for elements instead of types provided during instantiation. We believe that in our framework, this would just be a specific kind of conformance check, which can again be explicitly modelled and included for users.

VI. Conclusions

We have identified several shortcomings of current tools in how they handle the type/instance relation between models and type models. This relation is bidirectional, and consists of both the instantiation and conformance checking algorithm. While it holds some advantages, primarily in terms of usability and efficiency, hardcoding type/instance semantics in the core of the tool has a significant set of disadvantages that limit what can be done with the tool. To make matters worse, there is no common consensus on what the semantics of some of these concepts should be and how to handle violations. As this semantics is hardcoded in the tool, users are forced to use it, with the semantics only being defined in the source code of the tool.

We proposed a different approach, where both the instantiation and conformance checking algorithms, as well as the type mapping, is made explicit as a model. Explicit modelling of these semantics allowed us to clearly seperate model structure (a graph) from conformance semantics (an algorithm). Additionally, the mapping between different models also became available as a model that can be used like any other. The primary advantages of our approach are that users achieve full control over the modelling tool, and can themselves alter the semantics of what it means for a model to conform to a type model, or what it means to instantiate a type model. Previous approaches have limited themselves to providing users a pre-defined function which they had to deal with; deviations were not allowed. And even when extension was possible, this was limited to the implementation language of the tool, which might be unfamiliar to the user. In this stage of research, our approach has some problems with usability, as users are currently exposed to a great deal of complexity, to which they aren't exposed in other tools. We believe that this additional complexity can be handled through the use of more usable front-ends, which hide the complexity when it is not required. Performance problems can be resolved through efficient algorithms, similar to those found in other interpreters.

Our approach was illustrated with a small example that is prevalent in current programming languages: the semantics of subtyping. Many languages have their own semantics in this regard: is subtyping allowed at all? Do we use nominal or structural subtyping? What about multiple-inheritance? And what with the resolution order in multiple-inheritance? For each of these possible choices, there has been at least one programming language which choses one over the other. Their choices are often well-founded, making us believe that the ideal semantics is dependent on the domain. As the Modelverse intends to be a general (meta-)modelling tool, we should always offer the users the most appropriate formalism to describe their problem. When a problem naturally lends itself best to structural subtyping, for example, it should not be implemented using nominal subtyping.

In future work, we plan to address the usability problems we encountered in our tool, as well as performance.

Another direction for future work is the definition of a general type/instance relation, which holds for every model. That would make it possible to store each model as a conforming model (by construction), and to manipulate even incompatible models or non-strict models. Finally, the prospect of multiple type/instance relations, of which the semantics can be filled in by the user, raises many new research questions. Specifically, which current modelling operations can be lifted to this relation? For example, conformance between execution traces and an executable model, properties and models, and concrete syntax with abstract syntax. Similarly, can we then instantiate an executable model by giving an execution trace? Or automatically instantiate models with specific properties? All of these models have some relation between them, but is this really conformance, and what is the usefulness of identifying these relations as conformance relations?

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

[1] Colin Atkinson and Thomas Kühne. Concepts for comparing modeling tool architectures. In *Proceedings of MoDELS*, pages 398–413, 2005.

[2] Zoltan Theisz and Gergely Mezei. An algebraic instantiation technique illustrated by multilevel design patterns. In *Proceedings of MULTI*, 2015.

[3] Colin Atkinson and Thomas Kühne. Strict profiles: Why and how. In *Proceedings of UML*, pages 309–322, 2001.

[4] Francis Bordeleau. Model-based engineering: A new era based on Papyrus and open source tooling. In *Proceedings of OSS4MDE*, 2015.

[5] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *SIMULATION*, 86(2):109–126, 2009.

[6] Thomas Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5:369–385, 2006.

[7] Juan de Lara and Ester Guerra. Deep meta-modelling with MetaDepth. In *Proceedings of TOOLS*, pages 1–20, 2010.

[8] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations*, 2013.

[9] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Transactions on Software Engineering and Methodology*, 24(2):12:1–12:46, 2014.

[10] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. A unifying approach to connections for multi-level modeling. In *Proceedings of MoDELS*, 2015.

[11] Colin Atkinson, Bastian Kennel, and Björn Goß. Supporting constructive and exploratory modes of modeling in multi-level ontologies. In *Proceedings of Semantic Web Enabled Software Engineering*, pages 1–15, 2011.

[12] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[13] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A systematic approach to metamodeling environments and model transformation systems in VMTS. In *Proceedings of GraBaTs*, pages 65–75, 2004.

[14] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-based DSL frameworks. In *Proceedings of OOPSLA*, pages 602–615, 2006.

[15] Simon Van Mierlo, Bruno Barroca, Hans Vangheluwe, Eugene Syriani, and Thomas Kühne. Multi-level modelling in the modelverse. In *Proceedings of MULTI*, pages 83–92, 2014.

[16] Jesús Sánchez Cuadrado, Esther Guerra, and Juan Lara. Generic model transformations: Write once, reuse everywhere. In *Proceedings of ICMT*, pages 62–77, 2011.

[17] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. A-posteriori typing for model-driven engineering. In *Proceedings of MoDELS*, 2015.

[18] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[19] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of OOPSLA*, pages 318–326, 1997.

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

[21] Bruno Barroca, Thomas Kühne, and Hans Vangheluwe. Integrating language and ontology engineering. In *Proceedings of MPM: Multi-Paradigm Modelling Workshop*, pages 77–86, 2014.

[22] Ken Vanherpen, Joachim Denil, István Dávid, Paul De Meulenaere, Pieter J. Mosterman, Martin Törngren, Ahsan Qamar, and Hans Vangheluwe. Ontological reasoning for consistency in the design of cyber-physical systems. In *Proceedings of Cyber Physical Production Systems*, 2016. (under review).

[23] Yentl Van Tendeloo. Foundations of a multi-paradigm modelling tool. In *MoDELS ACM Student Research Competition*, pages 52–57, 2015.

[24] Pieter J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An introduction. *Simulation: Transactions of the Society for Modeling and Simulation International*, 80(9):433–450, 2004.

[25] Jaun de Lara and Esther Guerra. From types to type requirements: genericity for model-driven engineering. *Software & Systems Modeling*, 12(3):453–474, 2011.

[26] Louis Rose, Esther Guerra, Juan de Lara, Anne Etien, Dimitris Kolovos, and Richard Paige. Genericity for model management operations. *Software and Systems Modeling*, 12(1):201–219, 2011.

[27] Steffen Zschaler. Towards constraint-based model types: A generalised formal foundation for model genericity. In *Proceedings of VAO*, pages 11–18, 2014.

[28] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In *Proceedings of UML*, pages 290–304, 2004.

[29] Rick Salay, Marsha Chechik, Steve Easterbrook, Zinovy Diskin, Pete McCormick, Shiva Nejati, Mehrdad Sabetzadeh, and Petcharat Viriyakattiyaporn. An Eclipse-based tool framework for software model management. In *Proceedings of OOPSLA*, pages 55–59, 2007.

[30] Alessio Di Sandro, Rick Salay, Michalis Famelis, Sahar Kokaly, and Marsha Chechik. MMINT: A graphical tool for interactive model management. In *Proceedings of MoDELS*, 2015.

[31] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software and System Modeling*, 8:521–549, 2008.

[32] Clément Guy, Benoit Combemale, Steven Derrien, Jim Steel, and Jean-Marc Jézéquel. On model subtyping. In *Proceedings of ECMFA*, pages 400–415, 2012.

[33] Rick Salay and Marsha Chechik. Supporting agility in MDE through modeling language relaxation. In *Proceedings of the Workshop on Extreme Modelling*, pages 20–27, 2013.