On the Quest for Flexible Modelling

Esther Guerra
Universidad Autónoma de Madrid, Spain

Juan de Lara
Universidad Autónoma de Madrid, Spain

ABSTRACT

Modelling is a fundamental activity in Software Engineering, and central to model-based engineering approaches. It is used for different purposes, and so its nature can range from informal (e.g., as a casual mechanism for problem discussion and understanding) to fully formal (e.g., to enable the automated processing of models by model transformations). However, existing modelling tools only serve one of these two extreme purposes: either to create informal drawings or diagrams, or to build models fully conformant to their modelling language. This lack of reconciliation is hampering the adoption of model-based techniques in practice, as they are deemed too imprecise in the former case, and too rigid in the latter.

In this *new ideas* paper, we claim that modelling tools need further flexibility covering different stages, purposes and approaches to modelling. We detail requirements for such a new generation of modelling tools, describe our first steps towards their realization in the Kite meta-modelling tool, and showcase application scenarios.

CCS CONCEPTS

• **Software and its engineering → System modelling languages**; **Designing software**; *Design languages*;

KEYWORDS

Model-driven engineering; Flexible modelling; Modelling process

ACM Reference Format:
Esther Guerra and Juan de Lara. 2018. On the Quest for Flexible Modelling. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239376

1 INTRODUCTION

Modelling is pervasively used in Software Engineering to document, reason, design and understand software systems. Approaches like Model-Driven Engineering (MDE) [44] lift models as primary artefacts in the software development process, relying on them to specify, test, analyse, simulate and generate code for the targeted system. For this purpose, MDE approaches and tools require well-formed models, strictly conforming to the norms of the modelling language being used. However, some modelling scenarios would benefit from relaxing the conformance relation and tolerating inconsistencies. For example, early phases of model construction require informal discussions, which would become easier if the modelling environment is flexible with respect to conformance and inconsistency tolerance.

Most tools used for modelling support one extreme of the spectrum (informal modelling) or the other (fully formal modelling), with no assistance for a disciplined transition from informal to formal modelling. Moreover, the modelling process and the level of conformance required at each stage are hardly ever explicit. This makes assessing the current state of a modelling project difficult. We claim that this situation has hampered a wider adoption of MDE tools, which currently are positioned on the strict side and require models to be fully conformant to their meta-model all the time.

Moreover, the lack of flexibility of MDE frameworks adds unnecessary complexity to solutions targeting certain scenarios, like example-based modelling [34, 51], where models are created first, and used to derive the meta-model; meta-model unit testing [35], where unit tests may need to include ill-formed models; meta-model/model coevolution [23], as correct models may become ill-formed after their meta-model evolves, but the tool should still be able to load them; reuse of transformations [10], which typically can only be applied to models conforming to the meta-model the transformation was defined for; and multi-level modelling [6], as mainstream modelling supports two meta-levels only.

There is a growing interest in the MDE community to investigate ways to make modelling more flexible [21, 22, 24, 34, 42, 46]. However, nowadays, there is a lack of understanding on what are the requirements for flexible modelling tools, there is no systematic analysis of the scenarios where flexible modelling tools would bring advantages, and a unifying proposal for flexible modelling is missing. Despite some researchers proposing tools that provide certain flexibility to specific modelling tasks [22, 24, 34, 46, 52], they tend to implement partial solutions, lacking flexibility and customizability in aspects related to the offered meta-modelling facilities and the enactment of the modelling process.

To remedy this situation, in this *new ideas* paper, we identify scenarios that require flexibility in modelling and derive requirements from them. We have used these requirements to position existing flexible tools, and to design a proposal for a more flexible way of modelling and meta-modelling. Our proposal is distinctive in that we permit the customization of the conformance relation between a model and its meta-model to tolerate inconsistencies (e.g., violation of cardinality constraints, or objects with deferred typing). The stages in the modelling life-cycle of a project can be made explicit, together with heuristic quick fixes helping to transition from informal model states to more formal ones. The expressiveness of the meta-modelling environment is adjustable as well, e.g., enabling (or not) untyped objects, objects with multiple types, or multiple meta-levels. Finally, we support different meta-modelling processes, including 'types first' as in standard meta-modelling [47], 'models first' as in bottom-up and example-based approaches [11, 34], and co-creation of models and meta-models [22]. We show an initial

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239376

realization in the Kite tool, and validate the approach on several scenarios.

Paper organization. Section 2 argues on the need for flexible modelling, and Section 3 elaborates on requirements and scenarios. Next, Section 4 outlines the proposed architecture for flexible modelling, which is described in the following two sections: Section 5 introduces a meta-model for flexible modelling, and Section 6 reifies the modelling process as a model. Section 7 describes Kite, a prototype tool that gives support to these ideas. Section 8 exemplifies our approach on some application scenarios. Section 9 discusses related work and compares flexible tools based on the elicited requirements. Finally, Section 10 draws conclusions and lines for further research.

2 MOTIVATION

MDE advocates the automated manipulation of models, which requires having well-formed models compliant to the modelling language rules. Because of this, most MDE tools are built atop meta-modelling frameworks that make it easy to check the correctness of models, but where building incorrect ones may be tricky or not possible [47].

As an example, the Eclipse Modelling Framework (EMF) [47] – the de-facto standard infrastructure for domain-specific modelling used by many MDE tools – has little tolerance for inconsistencies. Ill-typed models (e.g., arising when the meta-model evolves, or needed in meta-model unit tests) cannot be loaded by the framework. In addition, EMF inherits some rigidities from its implementation language (Java), like immutable typing and two-level modelling.

However, some scenarios require building models that do not necessarily fulfil all conformance rules, such as at the initial stages of modelling or in meta-model unit testing [35]. Hence, we claim that a more flexible modelling infrastructure is needed, able to provide support for defining and enacting modelling and meta-modelling processes and their model conformance rules.

Modelling process. While models need to be fully-conformant to their meta-model to be tool-processable, their initial creation phases usually deal with partial or incomplete models. Partial models may include objects with no type (typing is deferred to a later stage), elements to support discussion but not intended to be part of the model, missing or extra slots and links, and may violate some cardinality or integrity constraints. Automated mechanisms to transition from informal models to more formal ones would be desirable. Moreover, different projects may need to follow different modelling processes.

Meta-modelling process. Modelling languages are typically created by defining their meta-model upfront, so that it can be instantiated to create models. While most tools follow this approach, other possibilities exist. For example, bottom-up meta-modelling [34] proposes creating models first, and then infer a meta-model. The rationale is that domain experts with low technical profile may find models easier to grasp than meta-models. Hence, using example models to drive the language development process promotes the active involvement of domain experts and contributes to improve the acceptance of the resulting language. Other proposals like [22] combine bottom-up and top-down approaches. A flexible meta-modelling infrastructure should support all these meta-modelling processes and provide facilities (primitives, refactorings, quick fixes...) tailored to each one of them.

3 REQUIREMENTS AND SCENARIOS

Based on the previous discussion, next we elicit requirements for a flexible modelling framework. These are also summarized in Figure 1 in the form of a feature model [27] (labels in attached circles refer to the requirement number addressing them). For each requirement, we also identify use cases and usage scenarios.

R1: Configurable inconsistency tolerance

In order to support both formal and informal modelling, the framework should allow relaxing the conformance relation and configuring the conformance rules to be checked on a model at any moment.

For instance, it should be possible to enable/disable the checking of cardinality and integrity constraints, or the correctness of feature values. If so configured, models may contain objects with an abstract type (e.g., to defer type concretization to a later stage), as well as objects with a non-existing type name (e.g., to evolve the meta-model with the missing type afterwards). Scenarios that would benefit from inconsistency tolerance include:

- *Model life cycle.* Consistency checking could be adjusted to be less strict in the initial modelling stages (e.g., for discussion) than in later stages (e.g., for model transformation or code generation).
- *Model migration, meta-model evolution.* Models can turn invalid if their meta-model changes. Still, if we relax the conformance relation, flexible modelling environments will be able to load them, becoming easier to fix [23].
- *Meta-model testing.* This scenario demands the creation of test models to validate a meta-model and detect flaws in its model acceptance and rejection capabilities. Some test models may focus on a particular aspect and therefore be incomplete. Other test models may need to be incorrect to allow assessing whether some meta-model constraint (e.g., a cardinality constraint) has been correctly captured and its violation issues an error [35, 43].
- *Test-driven meta-model development.* This scenario requires creating test models that refer to non-existing types and features, and the meta-model should be evolved either manually or via quick fixes to make it accept the models [34].

R2: Information extension

In standard modelling, objects are created using their classes as templates. This assigns a fixed set of slots to the objects, which cannot be extended later. However, enriching objects with additional information does not break type safety, but may be useful for informal modelling or as a preliminary step to extend their meta-model. Therefore, a flexible framework should allow configuring whether models can be extended with information not defined by the modelling language. This makes it possible to have objects with no type, as well as typed objects with slots and links not defined in the object's type. A fine-grained customization of these extensibility options would allow enabling or disabling untyped objects, and selecting the types whose instances can be extended.

This requirement enables the following application scenarios:

- *Data injection.* A typical approach to populate models from data is building a meta-model for the data technology (e.g., XML, JSON) and a model extractor [9]. Untyped objects may simplify this task, as no data technology meta-model is needed beforehand.

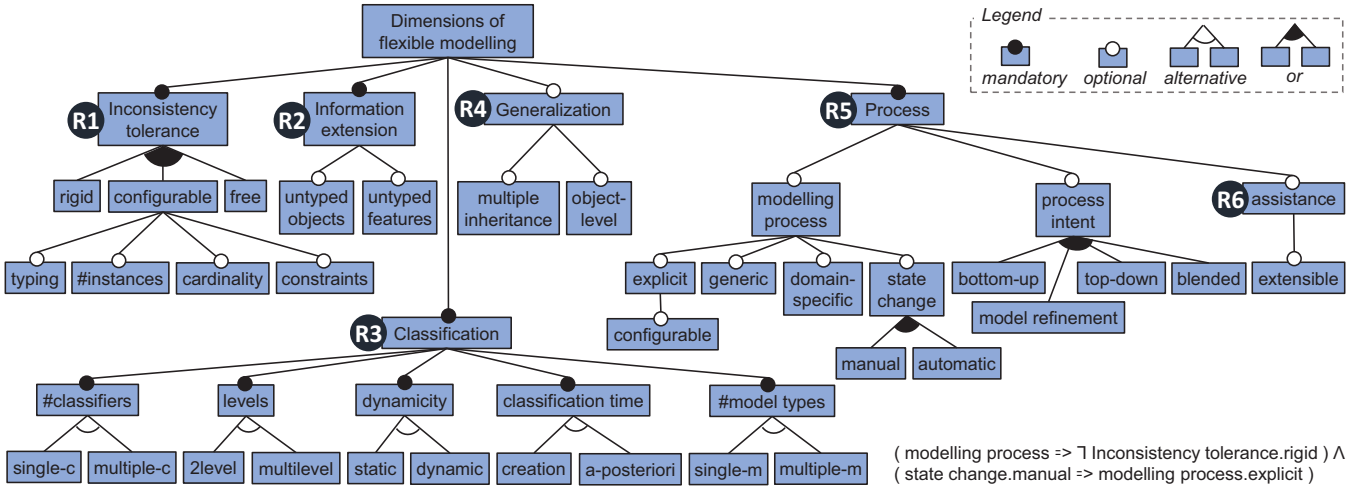


Figure 1: Requirements for flexible modelling. Feature Classification taken from [14].

- *Language extension and creation.* Unforeseen and emergent language features can be dynamically handled at the model level by adding new slots to objects, or representing them as untyped objects. This permits flexible extensions of existing languages in unforeseen ways, while in contrast, design-time language extension is based on class subtyping and has a static nature.
  Going a step further, untyped objects could trigger the creation of new types at the meta-model level, if they are deemed useful and general for the domain. This provides an iterative, example-based, bottom-up way to create languages [22, 34].
- *Auxiliary computation elements.* Some complex model manipulations require extending models with auxiliary constructs. For instance, calculating the transitive closure typically requires creating auxiliary edges [1], while simulation transformations may require auxiliary flags to mark the active elements [20]. Without proper support, developers need to extend the meta-model for each specific operation. While some model manipulation languages like EOL [30] are able to emulate dynamic object extension, making this feature native of the modelling framework would allow its usage for any transformation language.

R3: Configurable classification relation

Classification permits assigning types to instances [32]. The expressiveness of this relation is fixed in most modelling frameworks, and has the following limitations [14]: First, objects are classified by the class used to create them, and this classification cannot be changed afterwards (i.e., typing is static); second, objects and features are typically restricted to have exactly one type, despite some scenarios – like model operation reuse, see below – may be eased by supporting multiple typing; finally, standard meta-modelling frameworks only support two meta-levels, i.e., the classification relation cannot be iterated to obtain multiple meta-levels but this is emulated using workarounds like promotion transformations that convert objects into types [17]. These limitations not only affect the typing of objects, but also the typing of features.

Hence, to enhance flexibility, the framework should provide a more expressive classification relation supporting dynamic typing,

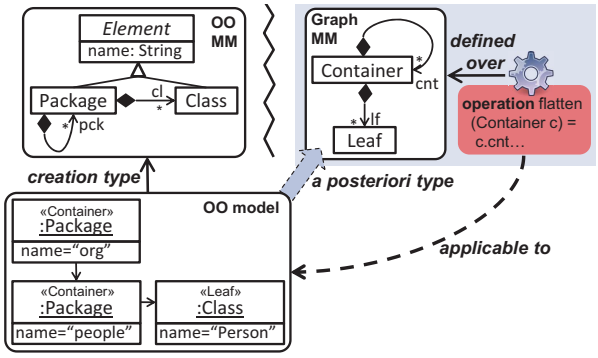


Figure 2: Model operation reuse by multiple typing.

multiple typing, multiple meta-levels, and the possibility to assign types to objects at creation time or *a posteriori*. Since not every application may require the full expressive power of the proposed classification relation, it should be customisable. For example, a certain scenario may require enabling/disabling more than two meta-levels, or restrict the set of classes an object may be typed by.

Some scenarios benefitting from this extra flexibility include:

- *Reuse of model operations*, such as model transformations and code generators. By allowing multiple typing, a model typed by a meta-model can be assigned types from other meta-models and reuse the operations defined for them. The typings assigned *a posteriori* could declare their own conformance rules [14].
  Figure 2 shows an example. An object-oriented model (in the lower-left) is assigned types from a graph meta-model (the extra types are displayed as object stereotypes). This way, the flatten operation, which is typed over the graph meta-model, can be applied as-is to the model.
- *Joint instantiation of sets of classes* and not only individual classes. This is useful when an object can take a combination of classes as type. The SMOF standard [39] reports this need with the Ontological Definition Meta-model (a meta-model for OWL [50]), where an instance can have several classifiers. For example, a property may be functional, transitive or symmetric (among others) or any

combinations of them. Supporting joint instantiation covers this scenario without blowing up the meta-model size with subclasses for every classifier combination.

- *Multi-level modelling.* Enabled by allowing multiple levels [6], this modelling paradigm is appropriate in scenarios that involve the type-object pattern, resulting in a layered organization of models with smaller size [8, 17]. Two domains that make pervasive use of this pattern are software architecture (where there are component types and instances, and port types and instances) and enterprise modelling (to represent task types and instances) [17]. Multi-level modelling is also useful for language extension [3] and domain-specific meta-modelling [16].

R4: Configurable generalisation relation

Generalisation is the creation of supertypes from subtypes [32]. Most meta-modelling frameworks support generalisation between types, which implies not only subtyping but also inheritance of the supertype features by the subtypes. A flexible framework should also permit generalisation between objects to enable inheritance of slot values [15], as well as customising whether multiple inheritance is allowed. An application of this requirement is the following:

- *Model libraries.* A way to achieve model reuse is by enabling inheritance at the object level. This opens the door to libraries of predefined models from which other models can inherit the structure [33], similar to prototype-instance languages [49].

R5: Explicit and configurable modelling process

The framework should permit defining modelling processes, their phases, and the conformance rules to be applied in each phase. A project should be able to configure its modelling process, or reuse one from a repository. A typical process might start accepting unconstrained models, with subsequent phases demanding stricter levels of conformance. Hence, this requires the possibility to configure the level of inconsistency tolerated (requirement R1). Processes may also dictate the creation order of the model elements according to their type. There should be mechanisms to determine the current model state, either manually through a "conformance" slider, or automatically based on the conformance rules the model fulfils and on the types of the objects it contains. Supporting this requirement allows defining customized modelling processes that guide the systematic construction and refinement of models.

Similarly, the process intent should be adjustable. For example, processes targeting the creation of meta-models may be top-down (types first), bottom-up (objects first) or blended, while other processes like model refinement focus on the evolution of models and the meta-model is not modified.

This requirement provides support for the following scenarios:

- *Transition from informal to formal modelling.* Models created in the initial phases are informal. Later, they may be refined, likely assisted by the environment, until obtaining full conformance.
- *Modelling guidelines.* Processes may serve as a guide for using a domain-specific language (DSL). This is especially useful for DSLs that are large or combine various diagram types (like UML or SysUML). For example, a UML project may typically (but not mandatorily) start by describing the classes, then focus on modelling their states, and later on collective interactions. Explicit processes can assist modellers in deciding the system aspect to focus on at each moment [36], and document the modelling practices within a project or company.

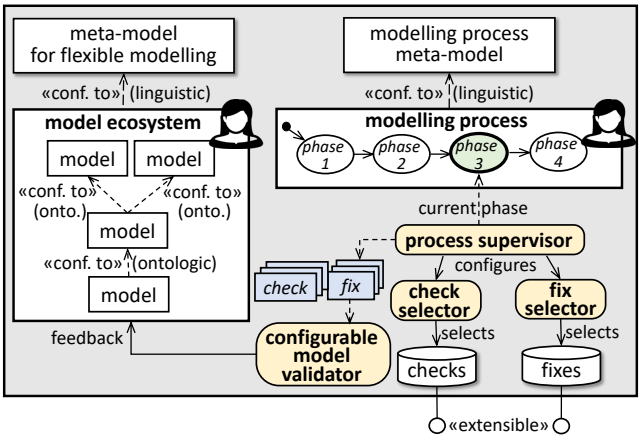


Figure 3: Proposed architecture for flexible modelling.

R6: Process-aware extensible assistance

Not only the level of conformance may change depending on the modelling phase, but the offered refactorings and quick fixes should be aware of the process: bottom-up fixes should modify the meta-model to repair disconformities in models, while top-down fixes should leave the meta-model untouched and modify just the models to correct detected issues. Moreover, domain-specific processes may introduce quick fixes tailored to a specific meta-model. Hence, the framework should be extensible with domain-specific conformance checks and fixes, which would integrate seamlessly with the predefined ones.

Scenarios benefitting from process-aware assistance include:

- *Model refinement.* Quick fixes in this scenario can help users to evolve informal models to conformant ones, and (domain-specific) refactorings can aid in obtaining higher quality models.
- *Bottom-up meta-modelling.* Quick fixes in bottom-up processes would assist in extending the meta-model based on non-conforming instance models [34].
- *Live meta-model/model co-evolution.* Changing a meta-model may impact its instance models. Quick fixes for breaking and resolvable changes [12] could be used to evolve the instance models upon meta-model updates.
- *Recommendation systems.* In line with current trends in recommender systems for software engineering [41], domain-specific processes may detect modelling idioms requiring assistance and propose fixes automating the completion of a modelling task [18].

4 PROPOSED ARCHITECTURE

We propose the architecture of Figure 3 to meet the previous requirements. It builds on two pillars: a flexible and expressive meta-modelling language, and a reification of the modelling process.

The proposed meta-modelling language adheres to the Orthogonal Classification Architecture (OCA) [7]. This distinguishes two kinds of typing: linguistic and ontological. The first one refers to the meta-modelling primitives used to create model elements. For

example, Ecore plays the role of linguistic meta-model in EMF, as it defines the primitives (EClass, EReference, etc.) to create elements in models. This way, a class Person in EMF is created by instantiating the linguistic type EClass. Ontological typing refers to the classification of objects by types within a domain. For example, Person is the ontological type of Einstein.

Our meta-modelling language defines a linguistic meta-model that reifies the ontological typing as an association. This enables a more flexible ontological typing that supports untyped elements, multiple typing, re-typing, and multi-level modelling. Moreover, our architecture permits adjusting the kind of conformance checks to perform on a model. Section 5 will describe our meta-modelling language for flexible modelling.

The second key element of the proposed architecture is support for explicit modelling processes. Specifically, it is possible to define processes and their intent, while the modelling phases may determine the conformance checks to perform. The system also provides a library of quick fixes which are selected depending on the inconsistency found and the process intent. For example, if there is a discrepancy between the type of an attribute (e.g., Integer) and its value (e.g., String), a model refinement process would propose a quick fix that changes the attribute value, while a process whose goal is creating a meta-model bottom-up would suggest changing the attribute type instead. While we propose some predefined processes for modelling and meta-modelling, we foresee the creation of domain- or project-specific processes, e.g., by means of extension points. Hence, both the modelling processes and the quick fixes can be extended by users. Section 6 will detail the handling of modelling processes.

5 META-MODEL FOR FLEXIBLE MODELLING

This section describes the linguistic meta-model for flexible meta-modelling. Section 5.1 presents the basic meta-modelling constructs and Section 5.2 explains the flexible typing facilities.

5.1 Basic modelling elements

Our meta-modelling language considers three kinds of modelling elements (see Figure 4): Models, Objects, and Features. The latter can be either Attributes or References.

The meta-model unifies the concepts of model and (meta-*)model (meta-class Model), class and object (meta-class Object), attribute and slot (meta-class Attribute), and association and link (meta-class Reference). This provides independence on the number of meta-levels, making it a level-agnostic meta-modelling language [5] and contributing to requirement R3. In contrast, other approaches, like UML [40], distinguish Classes from InstanceSpecifications (objects), hence being limited to two levels.

In our proposal, both Models and Objects can define Features and IntegrityConstraints. Instead, frameworks like EMF lack an explicit notion of model, requiring an extra root class – effectively playing the role of a model – holding global features and global constraints.

The meta-model permits defining *instantiation cardinalities* for Models, Objects and Features, which is controlled by attributes lbound and ubound inherited from TypedElement. This governs the number of instances of an element, and it allows, e.g., defining singleton objects (cardinality [1..1]) and abstract objects (cardinality [0..0]).

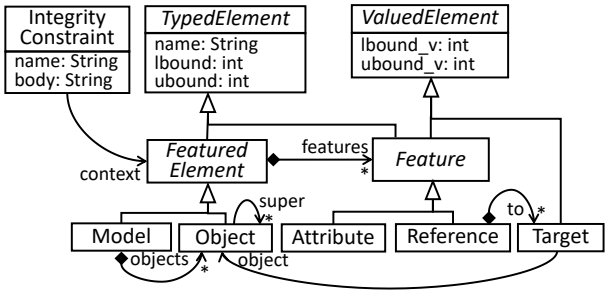


Figure 4: Basic elements for flexible modelling.

In addition, Features have a *value cardinality* which controls the number of allowed values (attributes lbound_v and ubound_v in meta-class ValuedElement). Modelling frameworks typically consider this kind of cardinality for features, but not instantiation cardinality. This way, our approach permits defining the attribute *name* with instantiation cardinality [0..*] and value cardinality [1..1], and then creating two instances (or slots) of it in an object: *name* with value "Peter", and *alias* with value "Ace". As this example also shows, the meta-model permits feature instances to use more specific names (*alias*) than their type (*name*). This is due to our more flexible notion of typing, which the next subsection will present.

Regarding References, they can point to several Target objects. Instead, most meta-modelling frameworks support this possibility at the model level (i.e., for links) but not in meta-models (i.e., for associations), and hence, if a reference needs to point to several classes, a common superclass for them must be introduced. Please note that our references are still binary relations. As our meta-model decouples instantiation and value cardinality in features, references need an additional class Target to hold the latter cardinality. This permits a fine-grain specification of cardinalities on the compatible objects of a reference. Another advantage of separating both cardinalities is making the instantiation of Models, Objects and Features uniform.

Generalization is defined over Objects, and hence applicable at every meta-level (requirement R4). While multiple-inheritance is the default, our framework can restrict it to single inheritance, as we will see in Section 6.

5.2 Flexible typing

In order to enable more flexible typings, our meta-model makes the typing relation explicit (meta-class Typing and its subclasses in Figure 5). The meta-model reifies the typing of models, objects and features. The typing of objects and models contains the typing of their owned features, and the typing of models contains in addition the typing of the model objects. Each typing relation is binary between an element playing the role of type, and another playing the role of instance. Differently from standard approaches, we allow zero, one or more typing relations stemming from the same instance.

Example 1. Figure 6 shows an instance of our linguistic meta-model, the upper part in abstract syntax, and the lower part using a textual concrete syntax that we have designed for our tool and will be used in the remainder of the paper. The figure depicts the typing relation between two models named Conference and MODELS. The former model defines objects Author and Reviewer, the latter of which must have at least one instance (lbound=1 and ubound=-1 in the

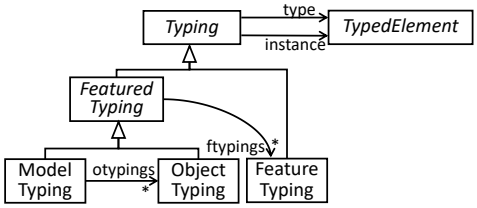
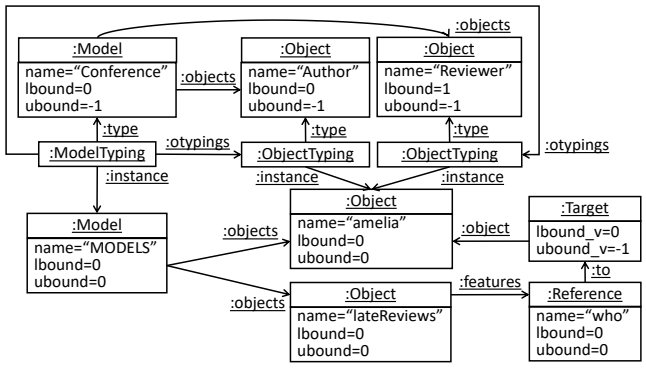


Figure 5: Meta-model for flexible modelling: Typing.

abstract syntax, represented by /1..*/ in line 3 of concrete syntax). For illustrative purposes, the MODELS model defines cardinality /0..0/ to enforce that it cannot be instantiated. This model defines two objects: amelia with two types, and lateReviews with no type. In this example, objects in model MODELS may be classified as Authors (e.g., if they have submitted a paper), Reviewers (e.g., if they have reviewed papers), both of them, or none. Moreover, the model includes information not foreseen by the Conference meta-model in the form of untyped objects (lateReviews) and features (who).




```
1  Conference {
2    Author {}
3    Reviewer /1..*/ {}
4  }
5
6  MODELS :Conference /0..0/ {
7    amelia :Author :Reviewer {}
8    lateReviews {
9      ref who = amelia;
10   }
11 }
```

Figure 6: Example of flexible typing in abstract (top) and concrete (down) textual syntax.

Representing the typing relation explicitly provides flexibility on the classification time, as the typing of an element can be assigned when it is created or later (requirement R3). As Example 1 illustrates, our approach enables elements (models, objects, features) to have no type (requirement R2) or multiple types (requirement R3). This type can be changed while preserving the identity of the instance. In contrast, most frameworks (e.g., EMF) delegate the semantics of typing to the implementation programming language (e.g., Java) and objects can only have one type, assigned at creation time.

Example 2. Listing 1 illustrates the usefulness of features with multiple types. Lines 1–6 declare model ArtistTypes, which contains an object Singer with attributes name and stageName. Lines 8–16

```
1  ArtistTypes {
2    Singer {
3      att name : String;
4      att stageName : String;
5    }
6  }
7
8  SomeMusicians :ArtistTypes {
9    tina :Singer {
10     att name = "Anna Mae Bullock";
11     att stageName = "Tina Turner";
12   }
13   joaquin :Singer {
14     att realName (:name :stageName) = "Joaquin Pascual";
15   }
16 }
```

Listing 1: Example of multiple typing for features.

define a model with two instances of Singer: tina and joaquin. Object tina assigns a value to each attribute (the typing is resolved by equality of names). Instead, object joaquin defines just one attribute called realName typed by both name and stageName. The effect is that both name and stageName share the same slot and therefore the same value, and moreover, both will be synchronized without the need to write ad-hoc synchronization code for that purpose. Such a multi-typing is transparent to model management operations, as traversing a collection of Singer objects and accessing name and stageName returns the value of realName in case of joaquin, and the values of name and stageName in case of tina.

Our meta-model unifies types and instances. The distinction between a class and an object lies in the role they play in the Typing relation (cf. Figure 5). Thus, our approach supports an arbitrary number of meta-levels, as an element playing the role of instance in a typing relation can play the role of type in another. For example, removing the annotation /0..0/ in line 6 of the listing in Figure 6 would allow the instantiation of model MODELS and its content.

The meta-model also contributes to requirement R1 (i.e., configurable inconsistency tolerance) by explicitly modelling all metamodelling facilities including typing, and not deferring their semantics to an external mechanism which cannot be changed (like a compilation into a programming language).

6 REIFYING THE MODELLING PROCESS

Our proposal permits making the modelling process explicit. To this end, it provides the meta-model in Figure 7 to define the phases of the modelling process and the transitions between them (requirement R5). Each phase sets the conformance checks it entails, and implicitly performs all checks defined in previous phases starting from the initial one. The checks may be selected among a set of predefined ConformanceRules (e.g., checking the correctness of types and feature values, or the satisfaction of integrity constraints) or be custom-made OclConditions. The TransitionMode between phases can be either manual (by the user) or automatic (when all checks in the current phase succeed). Moreover, transitions can define additional conditions for their completion, like the (non-)existence of certain modelling elements, supplying guidance for the modelling task.

Violations of a conformance check can trigger Quickfixes, which are filtered depending on the process Intent. Possible intents are: working on a model where the meta-model is fixed (MODELREFINEMENT), creating a meta-model (TOPDOWN), creating a model with

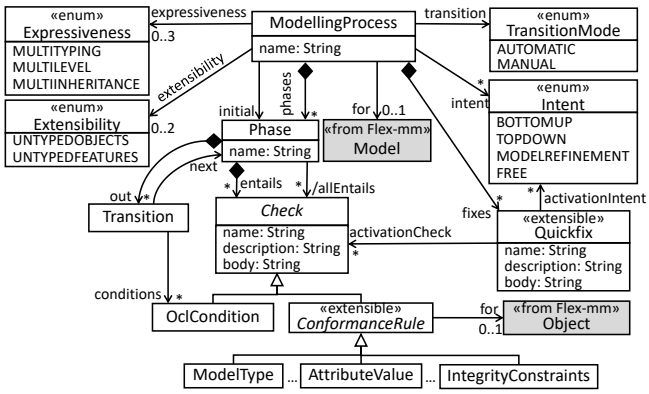


Figure 7: Meta-model for modelling processes.

the aim of growing a meta-model (BOTTOMUP), or free. This choice impacts the fixes offered. As an example, suppose an object declares a non-existing type; the quick fix for a BOTTOMUP process would add the type to the meta-model, while the one for MODELREFINEMENT would change the non-existing type by an existing one with similar name or features. In this way, we cover requirement R5.

Other configurable aspects of the modelling process are the Expressiveness of the modelling language (multiple typing, multiple meta-levels and multiple inheritance can be allowed or not) and whether Extensibility of models with untyped objects and features is permitted. It is possible to define generic modelling processes and domain-specific ones tailored to a meta-model (optional relation for in ModellingProcess). In the latter case, we can customize the conformance rules for specific types (relation for in ConformanceRule).

Two extension points (marked «extensible» in the meta-model) allow end-users to provide customized conformance rules and quick fixes which are handled as the predefined ones (requirement R6).

Example 3. Figure 8 shows a modelling process for generic model refinement. Its aim is providing a smooth transition from informal *draft* models to *strict* ones by gradually demanding stricter levels of conformance. The process defines five phases: *draft* (with no conformance checks), *typed* (the object types mentioned in the model must exist in the meta-model), *bounded* (the cardinality constraints must be satisfied), *well-formed* (the type of the feature values must be correct) and *strict* (the meta-model integrity constraints must be satisfied). Each phase also checks the conditions of previous phases. This way, the phases aggregate the conformance checks to be performed at each stage, and a model may be deemed conformant or not depending on the current phase, which in this case is selected manually. For extra flexibility, the process permits untyped objects and features, as well as multiple typing and multiple inheritance.

Although the defined phases are sequential, the progress towards a conformant model may not be sequential or even achieved at all. For instance, if the intention of a model is representing the results of an informal discussion, there is no need to reach strict conformance.

Figure 9 shows the conformance of a slight variant of the models in Figure 6 with respect to the process of Figure 8. If the user sets the current phase to Bounded, the checks validate the existence of type names (set by phase Typed) and the satisfaction of the instance and value cardinalities (set by phase Bounded). The checks in Typed fail because Comment does not exist in meta-model Conference. The

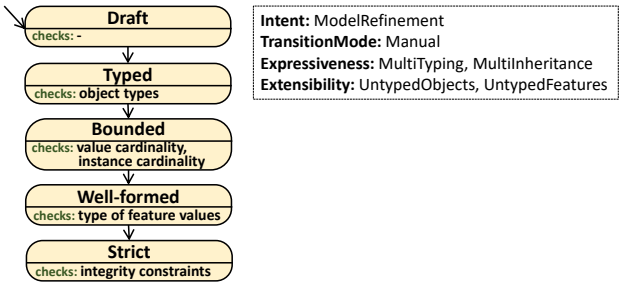


Figure 8: Generic model refinement process.

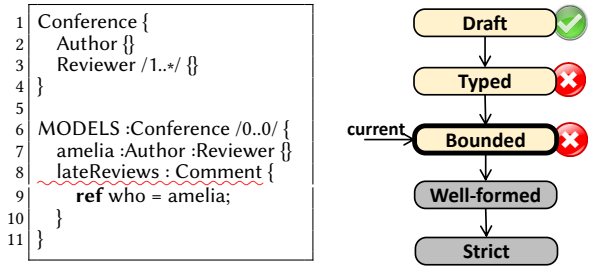


Figure 9: Conformance of a model with respect to a process.

model does not comply with phase Bounded because even if no cardinality is violated, the checks of Bounded include those of Typed. The checks for phases after the current one are not evaluated.

As the intent of the process is model refinement, the associated quick fixes change the model instance and leave the model type intact. In this example, the suggestions are either removing the type of object lateReviews, or changing it to Author or Reviewer. Both yield a valid model, as the process admits untyped objects (obtained by the first quick fix) and untyped features (obtained by the second).

7 PROTOTYPE IMPLEMENTATION

We have realized the previous ideas in a prototype tool called Kite. Kite is an Eclipse plugin based on EMF and Xtext. It supports textual modelling using the syntax shown in the listings, and uses the Epsilon Validation Language [31] (EVL) – a variant of OCL – as a constraint language. We opted to base our implementation on EMF because it facilitates its integration with many model management languages, like those of the Epsilon family [19] or ATL [26].

Figure 10 shows the tool in action. The editor (label 1) contains the definition of a simple goal modelling language (GoalML) and a goal model under construction (MobileAppReqs). The language integrity constraints to be evaluated on models are defined with EVL (label 2). For this purpose, we have implemented a model driver for Epsilon, able to understand the semantics of Kite.

The detected conformance errors and constraint violations are reported in the problems view (label 3). A properties page permits configuring the enabled conformance checks and the modelling process (label 4). The Modelling process section of the properties page permits choosing either no process, one of the standard processes we provide by default (model refinement, bottom-up or top-down), or the user can provide a domain-specific process (an instance of the meta-model in Figure 7). A process view shows the phases and transitions of the selected process, and permits the user to choose the phase with the desired conformity level (label 5).

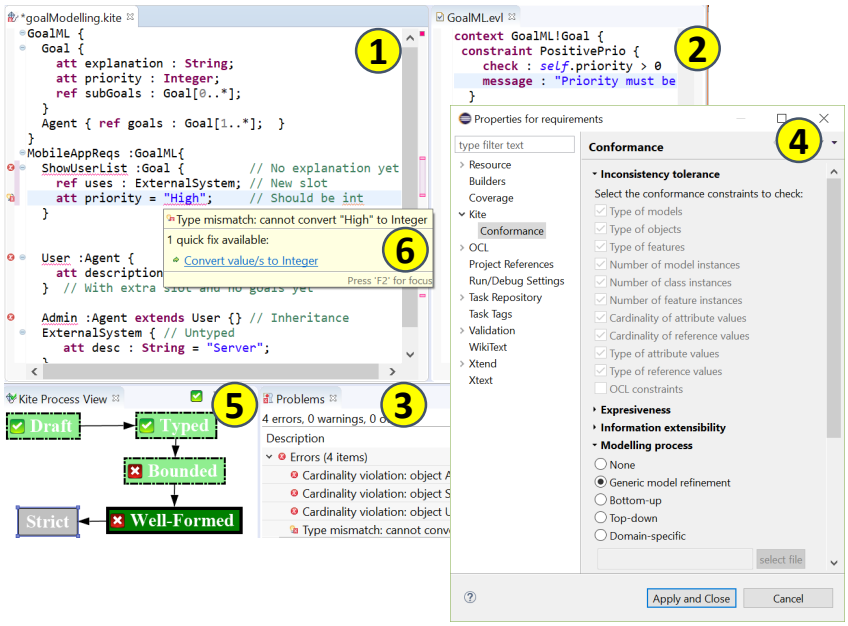


Figure 10: Using Kite in its IDE.

Building requirement models involves discussions in early modelling phases. This demands for flexible modelling before converging to stricter norms. Hence, the example follows a *model refinement* process. The user has manually selected phase *Well-Formed* as the current one (label 5). This and the previous phase yield errors because attribute priority has a wrong value, object ShowUserList misses an instance of feature explanation, and the instances of Agent lack goals. The integrity constraints are not evaluated at this stage, but at phase Strict. The process view shows this phase in grey (label 5) because it goes after the current phase. The model contains the untyped object ExternalSystem, which is allowed by the process.

The quick fixes offered by Kite (label 6) depend on the modelling process intent. In this example, the quick fix proposes converting an attribute value to Integer instead of changing the attribute type in the meta-model.

8 SHOWCASING SOME SCENARIOS

Next, we showcase some scenarios from Section 3, to validate the usefulness of our proposal and illustrate its benefits. For space limits, we focus on two of them: bottom-up modelling (Section 8.1) and model transformation reuse (Section 8.2). Along the paper, we also demonstrated explicit modelling processes (see Figs. 9 and 10).

8.1 Bottom-up meta-modelling

In bottom-up (or example-based) meta-modelling, meta-models are built driven by example models. This is an iterative process where models are constructed first, and then used to grow a meta-model with the types and features that the model contains.

Listing 2 shows an example model in the domain of education, inspired by [34]. The model may have been elicited in a brainstorming session with domain experts. It contains an object professor (prof1 in lines 2–5), a course (projectCourse in lines 6–10) and two groups (gr1 and gr2 in lines 11–12).

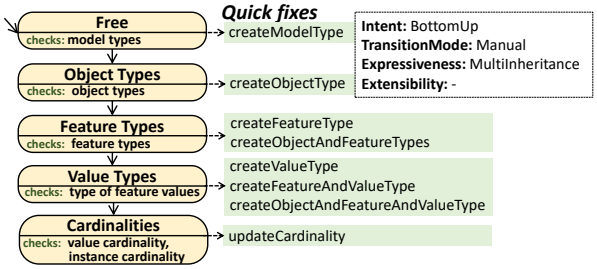


Figure 11: Bottom-up meta-modelling process.

```
1  aCourse :EduML {
2    prof1 {
3      att name = "Alan";
4      ref teaches = gr1;
5    }
6    projectCourse {
7      att name = "Sw Project";
8      ref coordinator = prof1;
9      ref groups = {gr1, gr2};
10   }
11   gr1 { att code = "2211"; }
12   gr2 { att code = "2212"; }
13
14 }
```

Listing 2: Example model.

```
1  EduML {
2    Professor {
3      att name;
4      ref teaches;
5    }
6    Course {
7      att name;
8      ref coordinator;
9      ref groups;
10   }
11   Group {
12     att code;
13   }
14 }
```

Listing 3: Inferred meta-model.

After collecting one or more example models, our approach can be used to build a meta-model assisted by quick fixes. Figure 11 shows our bottom-up generic process for this scenario. Successive phases set stricter conformance levels. At each stage, the suggested quick fixes modify the meta-model to reach the conformance level of the phase. This avoids committing to specific realizations of meta-model design decisions too early in the construction process.

The initial phase, Free, demands the existence of a meta-model. The quick fix available at this phase (createModelType) creates a meta-model for a model that is untyped or that refers to a non-existing

meta-model. In the former case, the quick fix asks for the name of the new meta-model. Phase Object Types is similar but for objects. Phase Feature Types checks for untyped features and offers quick fixes to create a feature type, or a class together with a feature type. Phase Value Types checks and offers fixes for the type of attribute and reference values. Finally, phase Cardinalities checks the number of instances of objects, features and feature values.

If we start from the example model in Listing 2, set the current phase to Feature Types, and apply all suggested quick fixes, we end up with the meta-model in Listing 3. As an example, applying the quick fix createObjectAndFeatureTypes to line 11 of the example model creates the class Group and its attribute code in the meta-model, while line 11 in the example model gets changed to gr1 :Group to reflect the new typing. The identification of types according to the feature values is deferred to stage Value Types, when further example models may have been provided. This may be useful to clarify, e.g., whether the attribute Group.code should be String or Integer. Note that Kite supports attributes with undefined data type, and references with undefined reference type.

Altogether, a flexible modelling framework like Kite can support bottom-up meta-modelling in a natural way. In contrast, standard meta-modelling frameworks like EMF do not support this way of meta-modelling directly. To overcome this rigidity, existing bottom-up tools either have created their own meta-model (including the typing relation) and offered services like our quick fixes (like in [34]), or have created a whole meta-modelling framework from scratch (like in [22]). Having a flexible modelling approach like ours as a starting point would have saved considerable effort.

8.2 Multiple typing for model operation reuse

In standard modelling approaches, model management operations (e.g., transformations) are typed according to a meta-model and cannot be reused for a different one. A way to overcome this problem and facilitate reuse is enabling multiple typing for objects [14].

```
1  GraphMM {
2    Container {
3      ref cnt : Container[0..*];
4      ref lf : Leaf[0..*];
5    }
6    Leaf {}
7  }
8  OOMM {
9    Element /0..0/ {
10     att name : String;
11   }
12   Package extends Element {
13     ref pck : Package[0..*];
14     ref cl : Class[0..*];
15   }
16   Class extends Element {}
17 }
```

Listing 4: Meta-models.

```
1  oomodel :OOMM :GraphMM {
2
3    pck1 :Package :Container {
4      att name = "org";
5      ref pck (:cnt) = pck2;
6    }
7
8    pck2 :Package :Container {
9      att name = "people";
10     ref cl (:lf) = person;
11   }
12
13   person :Class :Leaf {
14     att name = "Person";
15   }
16
17 }
```

Listing 5: Multi-typed model.

Listings 4 and 5 show how our approach can be used to support the model operation reuse example shown in Figure 2. Listing 4 declares a meta-model GraphMM over which a flatten operation has been defined elsewhere (lines 1–7), and a meta-model OOMM for object-oriented notations (lines 8–17). Model oomodel in Listing 5 is typed by both. In practice, a user may have created the model by instantiating OOMM, and additional types from GraphMM may have been added a posteriori. The system checks conformance to both typings equally regardless of their declaration order. Both objects and features can have multiple types. For instance, reference pck in line 5 is typed by Package.pck and Container.cnt.

Model management operations typed over either GraphMM or OOMM can be seamlessly applied over oomodel. For example, the query Container.allInstances() returns Sequence{pck1, pck2}, and query Package.allInstances() yields the same result.

New types for oomodel can be specified by hand (as we have done in this example), or they might be automatically added by so-called retyping specifications [14]. The latter are similar to model transformation rules that, upon matching an object, add a new type to it. Supporting retyping specifications in Kite is future work.

Altogether, standard modelling frameworks (like EMF) typically support single typing. Extending those frameworks for multi-typing-based reuse may be costly, while it is native in our approach.

9 RELATED WORK

Much research has been performed in tolerating inconsistency in software engineering [37], where some authors believe that *"[...] maintaining consistency at all times is counterproductive. In many cases, it may be desirable to tolerate or even encourage inconsistency, to prevent premature commitment to design decisions, and to ensure all stakeholder views are taken into account"* [38].

In contrast to this view, most standard MDE tools tend to favour or enforce consistency at all times. Our claim is that conformance inconsistency tolerance is needed to make MDE tools and processes more widely accepted. Since other researchers have also worked towards this vision, Table 1 reviews the position of flexible modelling tools using the requirements elicited in Section 3. The table characterizes tools as configurations of the feature model in Figure 1, and includes EMF as baseline. We target generic modelling tools and frameworks, i.e., we do not consider tools for specific scenarios like co-evolution. Next, we organize the comparison by tool goal, and also compare with related works in programming languages.

Flexible meta-modelling. A few proposals target a more flexible conformance relation. For instance, FlexiMeta [24] relaxes conformance to permit creating models and meta-models in any order. It is implemented in JavaScript, which permits prototyping objects with no defined class. A meta-model can be derived from the objects to enable simple checks (multiplicity violations, and missing or spare attributes). It implements an implicit modelling process for model refinement with three phases that can be selected manually: exploration (no meta-model), consolidation (meta-model with inconsistency tolerance), and finalization (meta-model and full conformance). JSMF [46] is a meta-modelling tool that permits dealing with incomplete and evolving requirements by enabling the (de-)activation of the cardinality and type checks of fields. It supports bottom-up and top-down meta-modelling as well as model refinement, but with no explicit modelling process. The flexible modelling tool FME [22] can also be used for a variety of intents (bottom-up, top-down, model refinement), but the modelling process is not explicit, and the conformance relation cannot be configured. In comparison, we support a finer-grain customization of the conformance rules and the expressiveness of the meta-modelling language; we allow defining processes making explicit the intent and phases of the modelling activity; and we provide quick fixes. Since we

Tool	Inconsistency tolerance (R1)	Information extension (R2)	Classification (R3)	Generalization (R4)	Process intent (R5.a)	Modelling process (R5.b)	Assistance (R6)
EMF [47]	rigid [a]	–	single-c, 2level, static creation, single-m	multiple inh.	top-down model refinement	–	–
FlexiMeta [24]	rigid, free	untyped objects (free) untyped feats (free)	single-c, 2level, static a-posteriori, single-m	–	model refinement	generic, manual	–
FlexiSketch [51]	free	untyped objects untyped feats	single-c, 2level, static a-posteriori, multiple-m	–	bottom-up model refinement	–	–
FME [22]	free	untyped objects untyped feats	single-c, 2level, static a-posteriori, single-m	multiple inh.	all	–	–
JSMF [46]	configurable	untyped feats	single-c, 2level, static creation, single-m	multiple inh.	all	–	–
Melanee [3, 4]	rigid	untyped objects untyped feats	single-c, multilevel, static creation, single-m	multiple inh. object-level	top-down model refinement	–	assistance
MetaBup [34]	free	untyped objects untyped feats	single-c, 2level, static creation, single-m	multiple inh.	bottom-up	–	extensible
MetaDepth [13, 14]	rigid	untyped objects untyped feats	multiple-c, multilevel, dynamic a-posteriori, multiple-m	multiple inh. object-level	top-down model refinement	–	–
ModelVerse [48]	rigid	–	single-c, 2level, static creation, single-m	multiple inh.	top-down model refinement	configurable, domain-specific automatic	–
Muddles [29]	free	untyped objects untyped feats	single-c, 2level, static a-posteriori, single-m	multiple inh.	bottom-up	–	–
Kite	configurable	untyped objects untyped feats	multiple-c, multilevel, dynamic a-posteriori, multiple-m	multiple inh. object-level	all	all	extensible

[a] Some non-conformant models cannot be loaded by the framework.

Table 1: Position of state-of-the-art tools in the flexible modelling design space with regards to the feature model in Figure 1.

model every meta-modelling facility, we are not constrained by the implementation language (e.g., JavaScript) which might hinder flexible features like multiple types or meta-levels.

Example-based modelling. Tools such as FlexiSketch [51], FME [22], metaBup [34] and Muddles [29] provide flexibility by facilitating the automatic or manual construction of a meta-model starting from a set of untyped models. Their modelling process is implicit, and while metaBup and Muddles only support bottom-up meta-modelling, the other tools support other process intents as well (see Table 1). None of these tools support explicit modelling processes, configurable conformance relations or process-aware quick fixes. Although metaBup provides an extension point to define domain-specific meta-model refactorings, they are unaware of the process.

Multi-level modelling. Another aspect of flexibility is the support for more than two meta-levels. There are a number of level-agnostic languages that fulfil this requirement, like MetaDepth [13, 14], Atkinson's [5], or Melanee [3], among others. Many of them offer control over information extensibility as well. However, they cannot configure the conformance relation, and lack support for modelling processes. Among these tools, only Melanee provides quick fixes, though they are hard-coded and cannot be extended [4].

Modelling processes. Few meta-modelling frameworks permit customising the modelling process. One of the exceptions is Modelverse [48], a tool for multi-paradigm modelling and simulation. It supports the explicit definition of mega-modelling processes, essentially model transformation chains between models likely conformant to different meta-models. In contrast, we aim at modelling the process of a model creation/evolution. More similar to our approach, Lúcio et al. [36] extend MPS with (Statechart-like) processes to guide the user in building a model. Transitions between states are performed when the model fulfils some condition. While we also support this scenario, our processes can tweak the conformance relation to achieve the required flexibility.

Programming languages. In programming languages, gradual typing [45] allows selecting which parts of a program are typed-checked statically, and which ones at run-time. The rationale is similar to ours: run-time checks permit rapid development, and static ones broaden error detection. In our case, an explicit process governs how to transition from more flexible to stricter typings.

Altogether, a significant number of tools aims at making modelling more flexible. However, they propose partial solutions where the conformance relation is fixed (except JSMF), or neglect the modelling process and assistance. The requirements we have elicited may help to improve these tools, and we encourage the community to explore this idea to increase the scope and acceptance of MDE.

10 CONCLUSIONS AND FUTURE WORK

This *new ideas* paper has argued on the needs for more flexibility in modelling, the benefits that it brings, and the technical requirements for flexible (meta-)modelling frameworks. We have proposed an architecture supporting configurable meta-modelling options and an explicit model of the modelling process. The architecture has been validated by an implementation in the Kite tool, and we have shown its benefits for flexible model refinement, example-based modelling, and transformation reuse.

We plan to continue working in Kite to reach a more mature state. This includes enlarging the set of quick fixes, improving how process models are defined, and integrating further model management languages. We are exploring ideas from meta-object protocols [28] to extend the framework semantics in a non-intrusive way, and allow building libraries of extensions of meta-modelling facilities like potency [6]. While this paper focused on making the conformance relation and the modelling process flexible, we have left out aspects of flexibility related to the concrete syntax. Our plan is to combine input models of multiple formats, such as textual, graphical and sketched. Finally, it would be useful to extend existing constraint languages and model reasoners to work with non-fully conformant models using ideas of paraconsistent logics [2, 25].

ACKNOWLEDGMENT

Work funded by the Spanish MINECO (TIN2014-52129-R) and the R&D programme of Madrid (S2013/ICE-3006).

REFERENCES

[1] Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai. 2005. Reusable idioms and patterns in graph transformation languages. *Electr. Notes Theor. Comput. Sci.* 127, 1 (2005), 181–192.

[2] Seiki Akama and Newton C. A. da Costa. 2016. Why paraconsistent logics? In *Towards Paraconsistent Engineering*. Intelligent Systems Reference Library, Vol. 110. Springer, 7–24.

[3] Colin Atkinson, Ralph Gerbig, and Mathias Fritzsche. 2015. A multi-level approach to modeling language extension in the enterprise systems domain. *Inf. Syst.* 54 (2015), 289–307.

[4] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. 2012. On-the-fly emendation of multi-level models. In *ECMFA (LNCS)*, Vol. 7349. Springer, 194–209.

[5] Colin Atkinson, Bastian Kennel, and Björn Goß. 2010. The level-agnostic modeling language. In *SLE (LNCS)*, Vol. 6563. Springer, 266–275.

[6] Colin Atkinson and Thomas Kühne. 2001. The essence of multilevel metamodeling. In *UML (LNCS)*, Vol. 2185. Springer, 19–33.

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

[8] Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *Software and System Modeling* 7, 3 (2008), 345–359.

[9] Jean Bézivin. 2005. Model driven engineering: An emerging technical space. In *GTTSE Revised Papers (LNCS)*, Vol. 4143. Springer, 36–64.

[10] Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2018. Model transformation reuse across metamodels: A classification and comparison of approaches. In *ICMT (LNCS)*, Vol. 10888. Springer, 92–109.

[11] Hyun Cho, Jeffrey G. Gray, and Eugene Syriani. 2012. Creating visual domain-specific modeling languages from end-user demonstration. In *MiSE @ ICSE*. 22–28.

[12] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *IEEE EDOC*. IEEE Computer Society, 222–231.

[13] Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with MetaDepth. In *TOOLS (LNCS)*, Vol. 6141. Springer, 1–20.

[14] Juan de Lara and Esther Guerra. 2017. *A posteriori* typing for model-driven engineering: Concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 31:1–31:60.

[15] Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. 2014. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.* 57, 1 (2014), 36–58.

[16] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2015. Model-driven engineering with domain-specific meta-modelling languages. *Software and Systems Modeling* 14, 1 (2015), 429–459.

[17] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46.

[18] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *MODELSWARD*. SciTePress, 282–290.

[19] Epsilon. 2012. http://www.eclipse.org/epsilon/.

[20] Claudia Ermel. 2006. *Simulation and animation of visual languages based on typed algebraic graph transformation.* Ph.D. Dissertation. TU Berlin. https://pdfs.semanticscholar.org/0dbf/38e3b2cc79f2b122adab82ca1f21e442942e.pdf.

[21] FlexMDE series of workshops. 2018. http://www.di.univaq.it/flexmde/index.php?pageId=previous_editions.

[22] Fahad Rafique Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. 2016. Using free modeling as an agile method for developing domain specific modeling languages. In *MoDELS*. ACM, 24–34.

[23] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. 2017. Approaches to co-evolution of metamodels and models: A survey. *IEEE Trans. Software Eng.* 43, 5 (2017), 396–414.

[24] Nicolas Hili. 2016. A metamodeling framework for promoting flexibility and creativity over strict model conformance. In *FlexMDE @ MoDELS*, Vol. 1694. CEUR, 2–11.

[25] Anthony Hunter and Bashar Nuseibeh. 1998. Managing inconsistent specifications: Reasoning, analysis, and action. *ACM Trans. Softw. Eng. Methodol.* 7, 4 (1998), 335–367.

[26] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Sci. Comp. Programming* 72, 1 (2008), 31–39.

[27] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. Carnegie-Mellon University Software Engineering Institute.

[28] Gregor Kiczales and Jim Des Rivieres. 1991. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA.

[29] Dimitrios S. Kolovos, Nicholas Drivalos Matragkas, Horacio Hoyos Rodriguez, and Richard F. Paige. 2013. Programmatic muddle management. In *XM @ MoDELS*, Vol. 1089. CEUR, 2–10.

[30] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. The Epsilon Object Language (EOL). In *ECMDA-FA'06 (LNCS)*, Vol. 4066. Springer, 128–142.

[31] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2009. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday (LNCS)*, Vol. 5115. Springer, 204–218.

[32] Thomas Kühne. 2009. Contrasting classification with generalisation. In *APCCM (CRPIT)*, Vol. 96. Australian Computer Society, 71–78.

[33] Zsolt Lattmann, Tamás Kecskés, Patrik Meijer, Gabor Karsai, Péter Völgyesi, and Ákos Lédeczi. 2016. Abstractions for modeling complex systems. In *ISoLA (II) (LNCS)*, Vol. 9953. 68–79.

[34] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Software and Systems Modeling* 14, 4 (2015), 1323–1347.

[35] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. 2016. Combining unit and specification-based testing for meta-model validation and verification. *Inf. Syst.* 62 (2016), 104–135.

[36] Levi Lúcio, Saad bin Abid, Salman Rahman, Vincent Aravantinos, Ralf Kuestner, and Eduard Harwardt. 2017. Process-aware model-driven development environments. In *MODELS Satellite Events (CEUR Workshop Proceedings)*, Vol. 2019. CEUR-WS.org, 405–411.

[37] Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. 2000. Leveraging inconsistency in software development. *IEEE Computer* 33, 4 (2000), 24–29.

[38] Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. 2001. Making inconsistency respectable in software development. *Journal of Systems and Software* 58, 2 (2001), 171–180.

[39] OMG. 2013. SMOF 1.0. http://www.omg.org/spec/SMOF/1.0/.

[40] OMG. 2017. UML 2.5.1. https://www.omg.org/spec/UML/.

[41] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. 2010. Recommendation systems for software engineering. *IEEE Software* 27, 4 (2010), 80–86.

[42] Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio. 2017. Special issue on Flexible Model Driven Engineering. *Computer Languages, Systems & Structures* 49 (2017), 174–175.

[43] Daniel A. Sadilek and Stephan Weißleder. 2008. Testing metamodels. In *ECMFA (LNCS)*, Vol. 5095. Springer, 294–309.

[44] Douglas C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer* 39, 2 (2006), 25–31.

[45] Jeremy G. Siek and Walid Taha. 2007. Gradual typing for objects. In *ECOOP (LNCS)*, Vol. 4609. Springer, 2–27.

[46] Jean-Sébastien Sottet and Nicolas Biri. 2016. JSMF: A javascript flexible modelling framework. In *FlexMDE @ MoDELS*, Vol. 1694. CEUR, 42–51.

[47] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition.* Addison-Wesley Professional, NJ.

[48] Yentl Van Tendeloo and Hans Vangheluwe. 2017. The modelverse: A tool for multi-paradigm modelling and simulation. In *WSC*. IEEE, 944–955.

[49] David M. Ungar and Randall B. Smith. 1991. SELF: The power of simplicity. *Lisp and Symbolic Computation* 4, 3 (1991), 187–205.

[50] W3C. 2017. OWL Web Ontology Language. https://www.w3.org/standards/techs/owl.

[51] Dustin Wüest, Norbert Seyff, and Martin Glinz. 2015. FLEXISKETCH TEAM: Collaborative sketching and notation creation on the fly. In *ICSE*, Vol. 2. IEEE, 685–688.

[52] Athanasios Zolotas, Robert Clarisó, Nicholas Matragkas, Dimitrios S. Kolovos, and Richard F. Paige. 2017. Constraint programming for type inference in flexible model-driven engineering. *Computer Languages, Systems & Structures* 49 (2017), 216–230.