MultEcore: Combining The Best of Fixed-Level and Multilevel Metamodelling

Fernando Macias[1,2], Adrian Rutle[1], and Volker Stolz[1,2]

[1] Bergen University College
first.last@hib.no
[2] University of Oslo

Abstract Mainstream metamodelling approaches based on the OMG standards, such as EMF, have a fixed number of modelling levels. Despite their partial acceptance in industry, limitations on the number of levels has led to problems such as lack of flexibility and mixed levels of abstraction. Existing multilevel modelling approaches have already tackled these problems by providing features like deep modelling and linguistic extension. However, these approaches have face challenges like hesitation of adoption by industry due to the lack of an ecosystem which frameworks like EMF already have. In this paper, we (1) propose a conceptual framework for multilevel modelling that tackles such challenges, (2) illustrate the approach with two examples, and (3) present the tool MultEcore, which makes the proposed framework directly compatible with EMF, combining the best of the two worlds of fixed- and multilevel metamodelling approaches.

1 Introduction

Model-Driven Software Engineering (MDSE) is a means for tackling the increasing complexity of software development processes through abstractions [7]. In MDSE, the main aspects of software are usually modelled using mainstream modelling approaches which conform to the OMG standards, such as Eclipse Modelling Framework (EMF) [22]. These approaches provide high reliability, a modelling ecosystem and good tool coverage. However, they are usually restricted to a fixed number of modelling levels, consequently suffering from mixed levels of abstraction, the need to encode a synthetic typing relation among model elements in the same level [14], and the difficulty to maintain and understand such models due to convolution.

Multilevel modelling addresses the issues of fixed-level metamodelling approaches by enabling an unlimited number of levels, deep hierarchies, potencies and linguistic extensions [4,9,19]. Employing a multilevel modelling stack has proven to be the best approach in many application scenarios [11]. Nevertheless, multilevel modelling also has several challenges hampering its wide-range adoption, such as lack of a clear consensus on fundamental concepts of the paradigm, which has in turn led to lack of common focus in current multilevel tools. Moreover, using multilevel approaches forces the use of specialised tools and technologies, causing technology lock-in. This restriction becomes an important disadvantage for integration and interoperability among existing modelling ecosystems.

In this paper, we introduce an approach which combines the best from the two worlds: fixed-level metamodelling with its mature tool ecosystem, and multilevel modelling with unlimited number of abstraction levels, potencies and linguistic extensions. Using our approach, model designers can seamlessly create a multilevel version of their hierarchies while still keeping all the advantages they get from fixed-level ones. In addition, we provide the following two new features for our multilevel approach, which we identified as important in modelling behaviour: (i) an ontological stack independent of any linguistic metamodel, and (ii) the idea of *(un)pluggable* linguistic metamodels onto the ontological modelling stack. We also present MultEcore, a tool for multilevel modelling without leaving the EMF world, and hence allowing reuse of existing EMF tools and plugins. As a proof of concept, we outline an application scenario in which we use MultEcore for runtime verification of behavioural models [16,17].

The outline of this paper is as follows: Section 2 examines existing multilevel modelling approaches and compares them to ours; Section 3 presents our conceptual framework, the tool MultEcore and two examples of its application; and Section 4 concludes the paper and outlines future lines of work.

2 Related Work

In this section we present some existing approaches that have tackled multilevel modelling by creating conceptual frameworks and tools. Generally, they focus on the definition of Domain Specific Modelling Languages (DSMLs), where the need for flexibility for both the typing relations and the numbers of levels is paramount. These approaches are based on the concept of *clabject* [1], which provides two facets for every modelling element, so that they can be seen as classes or as objects. Clabjects stem from the traditional object-oriented programming, so their realization into a metamodelling framework requires a linguistic metamodel that all the levels must share. That is, the clabject element is contained in a linguistic metamodel, together with other elements such as *field* and *constraint*.

MetaDepth [9] is a tool that allows to create deep metamodelling hierarchies. It serves as a proof of concept for a multilevel modelling framework that requires the aforementioned linguistic extension. MetaDepth supports several interesting features such as model transformation reuse [13] and generic metamodelling [10]. Melanee, which resembles MetaDepth, has been developed with a stronger focus on editing capabilities [2], as well as possible applications into the domains of executable models [3]. AtomPM [23] is a modelling framework highly focused on offering cloud and web tools. Modelverse [24], which is based on AtomPM, offers multilevel metamodelling functionalities by implementing the concept of clabject and building a linguistic metamodel that includes a synthetic typing relation. Finally, in [18], the authors assume that the only way to do multilevel metamodelling is by using the clabject approach. Therefore, they apply the same ideas for their implementation of multilevel modelling.

The aforementioned approaches implement their multilevel modelling solutions by using a linguistic metamodel including the clabject element, and “flattening” the ontological hierarchy as an instance of this linguistic metamodel. That is, the whole ontological stack becomes an instance of the clabject-based modelling language. Besides,

they require the creation of supporting tools from scratch, such as editors, constraint definition mechanisms and import/export capabilities to more widespread tools like EMF. We believe that multilevel modelling can be realised in a different way that improves its flexibility and removes the need for custom-made environments and tools. In this work, we present a different approach that does not require the definition of a specific linguistic metamodel nor a flattening of the ontological levels.

3 Conceptual Framework for Multilevel Modelling

In this section we explain the details of a conceptual multilevel modelling framework that realizes the concepts of potency and linguistic extension in a different way than the approaches listed in section 2. Then, we introduce MultEcore, a tool that enables multilevel metamodelling directly into conventional EMF. And finally, we discuss how this way of modelling in MultEcore can be applied into the domains of behavioural modelling and runtime verification [15].

The design of this conceptual framework is based on an ontological hierarchy of models with a fixed, common and generic topmost metamodel. In other words, the ontological hierarchy does not require a linguistic metamodel in order to be consistent, as opposed to the clabject-based proposals. The effect of this is a single, vertical hierarchy of models with a fixed topmost metamodel. Therefore, this hierarchy is similar to MOF, but does not restrict the number of new levels that the user can create. An overview of the proposed conceptual framework is displayed in Fig. 1. Note that since the ontological stack can grow downwards an arbitrary number of levels, these are indexed increasingly from the top. That is, the upper levels (with a higher degree of abstraction) have the lower indexes, as opposed to the standard OMG indexing.

In order to facilitate multilevel modelling directly in EMF, we have extended the idea of two-level cascading [5] (also referred to as "promotion transformation" [13]) in order to make it *repeatable* and *bidirectional*. Repeatable means that, instead of applying the two-level cascading just once, we allow to repeat it every time the user requires to create a new model as an instance of a previously existing instance, by transforming the latter into a model that can be instantiated. In EMF, this transformation implies generating an Ecore metamodel from an XMI instance.

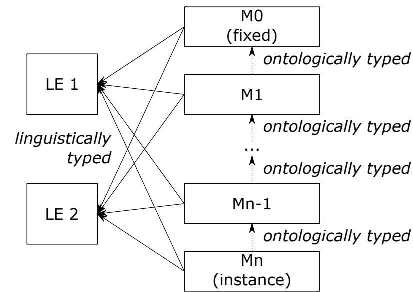


Figure 1: Overview of the multilevel conceptual framework

By itself, this repeatability only allows to generate new levels downwards. That is, create new instances with a lower level of abstraction. By making this process bidirectional, we allow to regenerate the instance version of a promoted model whenever it is required. In other words, in our extension of the two-level cascading, the process can be undone. This way, it is not required to store the information of the same model twice (as an instance and as a model). As explained in section 3.1, we use a level-agnostic representation for each level, from which both a model and an instance can be generated.

The process of applying two-level cascading in a repeatable and bidirectional way is illustrated in Fig. 2. This figure shows the ontological stack on the left, including the typing relation between models in two adjacent levels. Notice that the typing relation could also be labelled *instance of*. Therefore, we consider every model to be an instance of the model above, and we use the terms "model" and "instance" instead of "model" and "metamodel" to avoid confusion. As displayed, the topmost model in the ontological stack is `Ecore.ecore`, from which a user can create an Ecore model. The user can use any EMF editor configured with this second model to create an instance out of it (`m2.m1`, in XMI format). After creating the instance, EMF does not allow to create new levels on the stack, since an Ecore model is required for that.

Our contribution to overcome this limitation is the creation of a level-agnostic, graph-based representation of a model, independent of its level and typing relation. This representation can be generated from both an XMI instance and an Ecore model, and can be used to regenerate both of them. As a consequence, the user can get either the model or instance representation of a model in any level arbitrarily. That is, she gets a bidirectional application (upwards and downwards) of the two-level cascading technique. From now on, and based on the recurring dashed trapezium shape displayed on Fig. 2, we will refer to this technique as *sliding window*. Fig. 3 displays how our model representation is connected to the concept of sliding window by switching among the formats aforementioned. Moreover, the compatibility of our approach with conventional EMF allows to insert any pre-existing Ecore model into the ontological stack by just specifying the typing relations with the adjacent levels.

The left hand side of Fig. 1 shows two linguistic metamodels, namely `LE 1` and `LE 2`. We mentioned at the beginning of this section that the ontological stack in our framework does not require the existence of a linguistic metamodel. On the contrary, we propose using linguistic metamodels as a manner of adding new "views" to the models in the ontological stack. This way, the semantics of the models in the stack can be changed without modifying the models themselves, in a similar way as [12]. That is, any element from any model in the ontological stack can have one additional type (called classifier in some approaches) for each linguistic metamodel, and the number of linguistic metamodels is not restricted. This way, different and independent typing, all of them orthogonal to the ontological stack, can be *plugged* or *unplugged* from the stack without affecting its primary specification. In section 3.3 we illustrate how this usage of linguistic metamodel can be applied.

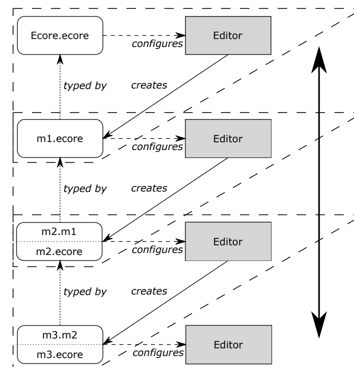


Figure 2: Repeated, bidirectional application of two-level cascading

The differentiating aspects of our conceptual framework are summarised as follows:

- A multilevel modelling stack that does not require linguistic metamodels, synthetic typing relations or "flattening" of the ontological stack.
- A realization of linguistic extension that differs from other approaches and allows for several, independent linguistic metamodels orthogonal to the ontological stack.

- Looser linguistic typing: while every element in every model has an ontological type, it does not require a linguistic type for each *plugged* linguistic metamodel.
- An extension of the two-level cascading approach that aides the implementation of the framework as an extension of EMF which preserves full compatibility with its model representations and tools.

According to [10], the use of a fixed topmost metamodel can pose a threat to the flexibility of the conceptual framework. The main criticism is that such a framework may be intrusive, since the user is forced to use the topmost metamodel. Hence, such a framework could not be used for scenarios in which the same model can have more than one "semantics", which implies that any element may have more than one type simultaneously. However, our conceptual framework can be used to overcome these issues, since the user can add a linguistic metamodel and add new types to the elements into the ontological stack. Furthermore, the possibility of adding new intermediate levels in a pre-existing stack favours the flexibility of switching or reusing metamodels. As explained before, this can be a solution for the issues presented in [12].

3.1 Tooling with MultEcore

One of the goals for the tooling aspect of our approach was keeping full compatibility with EMF. This way, the user does not depend on a set of "closed-garden" tools where the lack of proper documentation and maintainability is common. Moreover, the learning curve for the standard EMF user is almost non-existent. We realized the concept of sliding window in a small Eclipse plugin that offers users the possibility of switching between the Ecore and XMI representation of any model, so that she can still use her tools of choice. Fig. 2 shows how the sliding window concept applies to the EMF framework. In order to explain it, let us examine first how a basic modelling process is carried out in the EMF:

1. The user starts with the EMF reflexive metamodel (`Ecore.ecore`) at her disposal.
2. She uses an editor compatible with Ecore, which can be graphical, tree-based or textual, to create her own models (`M0.ecore` in the figure). These models will be typed by `Ecore.ecore`. The typing relationship means that the elements in this model are instances of Ecore elements, such as `EClass` and `ERelation`.
3. Once the user has created a custom model, she can use it to create a second editor.
4. The user can now build instances of the custom model `m1.ecore` using this second editor. The usual convention in EMF is that this instance has a file extension which is the same as the name of its model. Hence the name `m2.m1` in Fig. 2. However, this extension has no effect on the serialization of the instance, which always uses the XMI format.

At this point, the EMF hierarchy reaches its limit: it is not possible to repeat again the process of creating an editor from the bottommost model (XMI instance). In other words, it is not possible to create a new model at the level M3 typed by a model at the level M2. However, if the instance on the level M2 could be represented as a model (in Ecore format), the process of creating M3 could be carried out in the same way as the

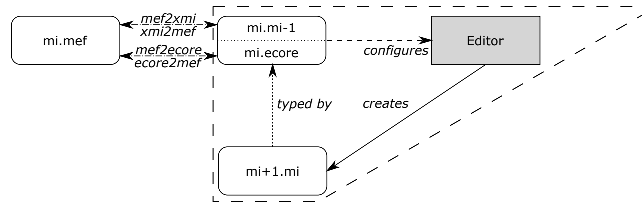


Figure 3: Implementation of the sliding window using MEF files

two last steps of the process described above. The sliding window allows the user to do exactly that. If the window slides down one level, the instance previously at the bottom of the window needs to be transformed into an Ecore model. On the contrary, sliding the window up one level requires to transform the model on top into an instance, and generating a model from the level above. This way, the user can (1) edit the instance at a level Mi and then *slide down* the window by switching to the model representation of Mi, which allows to create or edit the level Mi+1; or (2) get the instance representation of the current Ecore model to edit it, which implies *sliding up* the window.

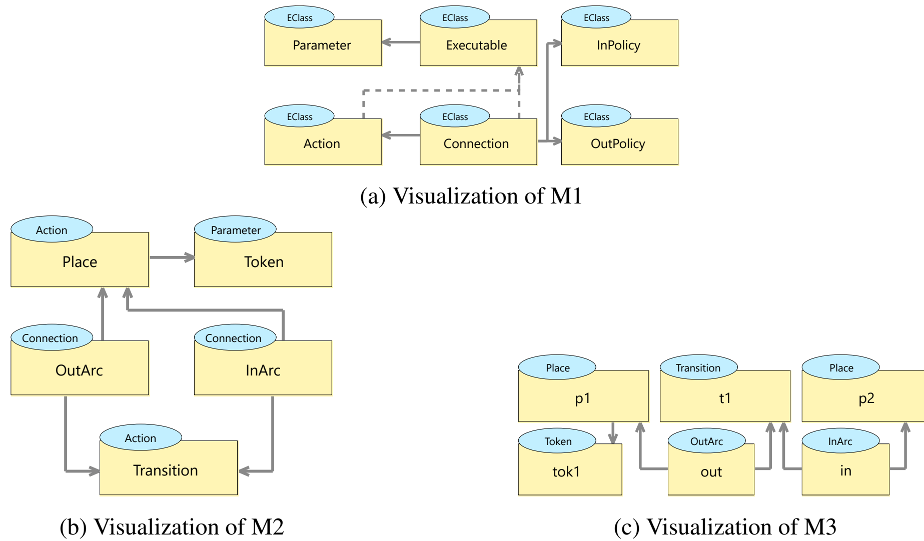
For a model in a given level Mi, the key to be able to switch between the model (`mi.ecore`) and the instance (`mi.mi-1`) representations is to have an intermediate, more abstract representation that is agnostic to the particularities of model and instance serialisations. Fig. 3 outlines how this intermediate representation is applied in the sliding window. In this abstract representation, models are considered to be attributed-graph based. Therefore, they only consist of *elements* and *relations* (equivalent to `EClass` and `EReference` in EMF). In addition, we store information such as the name, type, and potency of each element. The intermediate representation is serialised in XML format, following a particular schema called *MultEcore Format*. Consequently, we identify this files with the extension MEF.

Fig. 3 shows how MultEcore uses a MEF file (`mi.mef`) to enable switching between both representations on a level Mi. We defined two translations for each representation:

- *mef2xmi*, to get the instance representation of a level;
- *xmi2mef*, to reflect the changes from the instance into the MEF file;
- *mef2ecore*, to generate the model representation of a level; and
- *ecore2mef*, to reflect the changes from the model into the MEF file.

It is worth pointing out that the execution of *mef2xmi* on a level Mi triggers the execution of *mef2ecore* in the level Mi-1. The four translations can be executed in a user-transparent fashion, and their execution times for the example presented in section 3.2 are negligible. Even further, the process of translating back and forth among the three representations (Ecore, XMI and MEF) can be transparent to the user. It is trivial to hide the process of interacting with the MEF files, and simply offer the possibility to select one level and get the desired representation, using the *mef2x* transformations. This way, the user is only aware of the existence of several levels and the possibility of getting the model or the instance representations for each of them. Any change on the generated models is automatically reflected in the MEF files at the time of saving.

We have created a custom editor with Eclipse Sirius [21]. This editor (also called *viewpoint* in Sirius' terminology) simplifies the visualization and highlights multilevel



(a) Visualization of M1

(b) Visualization of M2

(c) Visualization of M3

Figure 4: Models created with MultEcore

features such as the type of an element. This editor allows to create instances of any ontological type, including Ecore types, metamodel types, and meta-metamodel types via potency, which are discovered by transitively visiting all the available supertypes. The visualisation of the example presented in section 3.2 was created using this editor.

Note that other multilevel-aware tools such as constraint and transformation languages would require to either flatten the whole ontological and linguistic hierarchies or reimplement them with multilevel capabilities. Nevertheless, existing ones such as ATL and OCL can be used seamlessly on the Ecore and XMI representations.

3.2 Application of Multilevel Modelling for Behavioural Modelling

Fig. 4 shows three levels of an example multilevel stack created with MultEcore. M1 defines the generic elements that appear in behavioural modelling languages such as activity diagrams, Petri nets and state machines. In the level M2 the language development takes place, and the metamodel of these languages must comply with the metamodel at M1. And in M3, a particular instance of the language specified in M2 is created. The topmost level M0 consists of the Ecore reflective metamodel, which is not displayed. The bottommost level M4 represents, in this example, the state of the system at a particular point during runtime, if the stack is used for executable modelling with model transformations as execution semantics. Such capabilities are still a work in progress, so M4 is also excluded from Fig. 4. Hence, only the levels M1 to M3 are displayed.

All the concepts in M1 are instances of `EClass` and `EReference`. This model consists of two main concepts `Action` and `Connection`. Both of them inherit from the abstract concept `Executable`, whose semantics is *something that runs*. `Action` represents the abstraction of a node, state or place. That is, the current task or state that the system is concerned about in a particular moment in time. The concept `Connection`

abstracts the idea of a transition from one `Action` to another. We consider that giving this transition the semantics of *something that runs* is required to allow the representation of some behavioural DSMLs such as UML sequence diagrams, in which the passing of a message might require the execution of operations and assignments. In these cases, a connection contains some executable semantics, rather than acting as a mere jump from one `Action` to another. Both `Action` and `Connection` inherit a relation with `Parameter`. This concept is used to represent passing of information at runtime. Notice that there are some implicit semantics in the way this class is related to `Executable`. By default, all parameters are of the input-output type, and can be connected to more than one element to indicate parameter passing or global variables. The attributes `InPolicy` and `OutPolicy` are used to enrich the behaviour of a given `Connection`. These are not relevant for this example and are omitted here.

M2 defines a classic Petri Net model, with `Place` and `Transition` as instances of `Action`, `Tokens` as parameters that can be passed between places, and arcs that connect places and transitions. M3 represents a simple Petri Net with two places `p1` and `p2`, an intermediate transition `t1`, the corresponding arcs, and a token `tok1` in `p1`. Note that this representations uses the abstract syntax from our tool. If required, the modelling engineer can define a new editor (e.g. in Sirius) with a more specific syntax.

3.3 Application of Linguistic Metamodel for Runtime Verification

The literature shows many examples of behavioural DSMLs created with multilevel modelling. Motivated by those examples, we used our conceptual framework to create a linguistic metamodel that enables runtime verification in any behavioural DSML specified in the ontological stack. This idea was originally presented in [16]. This linguistic metamodel enables the creation of temporal properties in any propositional logic. We have chosen the syntax and semantics of LTL for this illustrative example due to its simplicity and popularity, but it can be substituted by other suitable logics such as CTL [8] or languages like SALT [6].

Independently of the chosen language, there are two key aspects in the way we link, via linguistic metamodel, the formulae for temporal properties with the models in the ontological stack. The first aspect is considering atomic propositions as collections of elements of any given level in the ontological stack. The evaluation of an atomic proposition should yield `TRUE` or `FALSE`. This result is evaluated by matching the elements in the atomic proposition with those in the corresponding level of the ontological stack. The most common scenario is looking for a match of a particular set of element instances in the bottommost level, which represents the state of the modelled system at runtime, as hinted in section 3.2. This process is explained in more detail in [17]. The second key aspect is the translation of the grammar of the chosen language into a model with equivalent syntax and semantics. Fig. 5 shows this model for our LTL example.

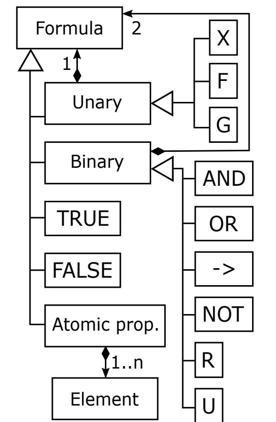


Figure 5: LTL metamodel

This linguistic metamodel consists of `Formulae` which are applied to `Models`, which contain a collection of model `Elements`. `Element` is the linguistic type that can be added to all elements, at any level, on the ontological stack. Besides the model contains the `Unary` operators `G`, `F` and `X` and the `Binary` operators `U` and `R`. The unary operators mean, respectively, *the formula must hold now and in the future*, *the formula must hold eventually* and *the formula must hold in the next point in time*. As for the binary ones, `U` is interpreted as *one formula must hold as long as the other one does not hold*, and `R` is interpreted as *one formula has to hold at least until the first time that the other one holds*. The logical operators `AND`, `OR` and `NOT` and the unidirectional implication arrow $\rightarrow$ complete the syntax.

As an example of a property modelled with this technique, Fig. 6 displays a simplified version of the liveness property for Petri Nets (see section 3.2) with just one `Transition t` between two `Places x` and `y`. Notice that the `Elements` from the ontological hierarchy and their relations are depicted with a dotted border, and that they act as variables. This means that their names are used to identify the same element in two different points in time. For example, `x` represents the same instance of a particular place in two different states of the system, one before passing the token and one after. Additionally, the variable name can be omitted if it is not necessary.

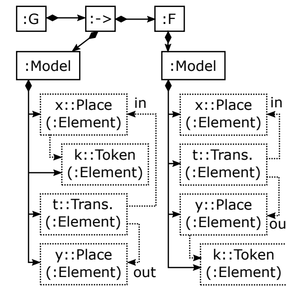


Figure 6: Example of the liveness temporal property

The liveness property holds if every time that the input place (`x`) of a transition (`t`) has a token (`k`), the transition is fired, passing the token to the output place (`y`) of that transition. Other safety and co-safety properties can be modelled in a similar fashion.

4 Conclusions and Future Work

We have presented a conceptual framework that enables multilevel modelling in an alternative way, compared to the state of the art. This paper shows how the approach can be applied into commonly researched scenarios, such as behavioural modelling, as well as less common ones like runtime verification. We have presented the tool MultEcore that implements this conceptual framework directly in EMF. This implementation reduces the learning curve, eliminates technology lock-in and provides full compatibility with EMF's set of tools and techniques. Our prototype Eclipse-based implementation with the example on behaviour modelling through Petri Nets is available from `http://prosjekt.hib.no/ict/multecore/`.

The next steps of this work are the extension of the tool functionalities, such as querying and navigation of subtypes and supertypes. We also intend to use MultEcore to create a hierarchy of behavioural models. Afterwards, we plan to define semantics for that hierarchy of languages using *coupled model transformations* [20] exploiting the existence of several levels of abstraction. These transformations will configure an execution engine for the behavioural models in the ontological stack. Finally, we will formalize the MEF representation and the related transformations as a model and model transformations, respectively.

References

1. C. Atkinson. Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop (EDOC'97)*, pages 90–101. IEEE, 1997.
2. C. Atkinson and R. Gerbig. Flexible deep modeling with Melanee. In *Modellierung (Workshops)*, volume 255 of *LNI*, pages 117–122. GI, 2016.
3. C. Atkinson, R. Gerbig, and N. Metzger. On the execution of deep models. In *1st Intl. Workshop on Executable Modeling*, volume 1560 of *CEUR Workshop Proceedings*, 2015.
4. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.
5. C. Atkinson and T. Kühne. Concepts for comparing modeling tool architectures. In *MoDELS*, volume 3713 of *LNCS*, pages 398–413. Springer, 2005.
6. A. Bauer, M. Leucker, and J. Streit. SALT—structured assertion language for temporal logic. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *LNCS*, pages 757–775. Springer, 2006.
7. M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.
8. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
9. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *Objects, Models, Components, Patterns*, volume 6141 of *LNCS*. Springer, 2010.
10. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS*, volume 6394 of *LNCS*. Springer, 2010.
11. J. de Lara, E. Guerra, and J. S. Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, Dec. 2014.
12. J. de Lara, E. Guerra, and J. S. Cuadrado. A-posteriori typing for model-driven engineering. In *MoDELS*, pages 156–165. IEEE, 2015.
13. J. de Lara, E. Guerra, and J. S. Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015.
14. T. Kühne and D. Schreiber. Can programming be liberated from the two-level style: multi-level programming with DeepJava. *ACM SIGPLAN Notices*, 42(10):229–244, 2007.
15. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
16. F. Macias, A. Rutle, and V. Stolz. A Property Specification Language for Runtime Verification of Executable Models. In *NWPT*, pages 97–99, 2015. Technical report RUTR-SCS16001, School of Computer Science, Reykjavik University.
17. F. Macias, T. Scheffel, M. Schmitz, and R. Wang. Integration of runtime verification into meta-modeling for simulation and code generation. In *Intl. Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*. Springer, 2016.
18. F. Mallet, F. Lagarde, C. André, S. Gérard, and F. Terrier. An automated process for implementing multilevel domain models. In M. van den Brand, D. Gašević, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *LNCS*, pages 314–333. Springer, 2010.
19. A. Rossini, J. de Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014.
20. A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Behaviour Modelling — Foundations and Applications*. ACM, 2012.
21. Sirius. *Project Web Site*. `https://eclipse.org/sirius/` Accessed 21 of July 2016.
22. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2008.
23. E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. AToMPM: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, 2013.
24. S. Van Mierlo, B. Barroca, H. Vangheluwe, E. Syriani, and T. Kühne. Multi-level modelling in the Modelverse. In *MULTI@ MoDELS*, volume 1286 of *CEUR Workshop Proceedings*, 2014.