Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Executing Multilevel Domain-Specific Models in Maude

Alejandro Rodríguez[a] Francisco Durán[b] Adrian Rutle[a]
Lars Michael Kristensen[a]

a. Western Norway University of Applied Sciences, Bergen, Norway

b. Universidad de Málaga, Málaga, Spain

Abstract Multilevel modelling (MLM) tackles the limitation in the number of abstraction levels present in traditional modelling approaches within the model-driven software engineering (MDSE) field. One way to specify the behaviour description of MLMs is by means of multilevel model transformations. In this paper, we propose an approach to achieve reusability and flexibility in specifying and executing multilevel model transformations. For this purpose, we rely on code-generation and the efficient rewriting logic mechanisms that Maude provides. As a proof of concept, we have developed an infrastructure which combines our MLM tool MultEcore, that facilitates definition of MLM hierarchies and transformations, with Maude, which performs the execution of the transformations on these hierarchies.

Keywords Multilevel modelling; Model transformations; Rewriting logic

1 Introduction

MDSE tackles the increasing complexity of software by utilizing abstractions and modelling techniques, and treats models as first-class entities in all phases of software development. MDSE has proven to be a successful approach in terms of gaining quality and efficiency [WHR14, MGS+13]. Most traditional MDSE approaches are based on the Object Management Group (OMG) 4-layer architecture, such as the Eclipse Modelling Framework (EMF) [SBMP08] and the Unified Modelling Language (UML) [UML]. These approaches follow a two-level hierarchy in which only two levels of abstraction are available for the modeller; i.e., models and their instances. Compelling to use these two-level (meta)modelling approaches may introduce several challenges, for instance, convolution and an increase in the complexity of models [LGC14, LG18]. It also has a direct impact in the specification of Domain-Specific Modelling Languages (DSML), since the domain expert might be forced to fit several abstraction layers into the only two levels which are supported by the traditional approaches [AK08]. Furthermore, capturing all the concepts in the same level makes it more difficult to define the metamodel and to fix the potential inconsistencies created in the artefacts conforming to (or depending on) this metamodel.

Alejandro Rodríguez, Francisco Durán, Adrian Rutle, Lars Michael Kristensen. *Executing Multilevel Domain-Specific Models in Maude.* Licensed under Attribution 4.0 International (CC BY 4.0). In Journal of Object Technology, vol. 18, no. 2, 2019, pages 4:1–21. doi:10.5381/jot.2019.18.2.a4

MLM has proven to be a successful approach in areas such as software architecture and enterprise/process modelling domains [LGC14, AK17, AKdL18]. Having a hierarchical organization of the metamodels defined to precisely capture the desired environment facilitates the possible extensions and modifications that might come in the future, not only in the existing levels, but also for adding/removing levels. MLM provides separation of concerns and therefore prevent the pollution of models where specialization of concepts would be done in the same level. This also leads to a better modularization and facilitates extendibility. Being able to add new metalevels makes extensions/modifications independent on other models. Further benefits of MLM and a detailed comparison between MLM and two-level traditional approaches can be found in [LG18].

Understanding the behaviour of a model is key to comprehend the behaviour of the underlying system that is being abstracted. In MDSE, model transformations are one of the possible means to specify behaviour. Although there are several approaches proposed for the definition and simulation of behavioural models based on reusable model transformations (e.g., [dLV02, Ren03, RDV09]), these rely on traditional two-level modelling hierarchies. Furthermore, modelling the behaviour through multilevel model transformations [AGM15] and performing execution in MLM has not been widely explored yet. Multilevel Coupled Model Transformations (MCMTs) have already been proposed [MRS+18b, MWR+19] to achieve reusable multilevel model transformations for the definition of behaviour. In this paper, we have improved the MCMTs by making them more reusable and flexible, extended them with the notion of cardinality, and implemented a first prototype for the execution of the rules.

In this paper, we propose an infrastructure for the execution of MLM hierarchies. This infrastructure is built on top of previous work for specification of structure and behaviour of MLM hierarchies in MultEcore [MRS16, MWR+19, MRS+18b]. MultEcore is a set of Eclipse plugins aimed to combine the best from traditional two-level modelling – the mature tool ecosystem (integration with EMF) and familiarity – with the flexibility of MLM. It supports the main features that characterize MLM such as potency, multiple typing and unlimited level of abstractions.

We rely on Maude for the execution/simulation of MLM hierarchies [CDE+07]. Maude is a high-level language and a high-performance interpreter and compiler in the OBJ algebraic specification family [GM13]. It supports rewriting logic and programming of systems. Among the functionalities that Maude provides, we exploit the ability to specify object-based systems which allows us to transform both the multilevel hierarchy and the MCMTs from MultEcore to Maude. This transformation provides the complete Maude specification (a rewrite logic theory) that can be directly executed by the rewriting logic engine. Execution in Maude means to apply the rewrite rules that gives the next states of our model. Ultimately, we can conduct reachability analysis (by means of strategies [EMOMV07]) and model checking. Maude supports model checking on the generated state space as it implements a Linear Temporal Logic (LTL) [BK08] model checker.

Paper outline: We present the prototype infrastructure in Sect. 2. Section 3 introduces the MLM background and the running example which we use for the rest of the paper. In Sect. 4 we describe how MCMTs work and display the rules we define for the MLM hierarchy example presented in Sect. 3. Section 5 discusses how Maude can be used to execute the MLM hierarchy and how the translation between MultEcore and Maude has been achieved. In Section 6 we discuss related work. Finally, Section 7 concludes the paper and outlines directions for future work.

2 The infrastructure

In this section, we present the overall architecture (see Fig. 1) of the infrastructure which we have developed for the execution of MLM hierarchies. The left-hand side of Fig. 1 shows the MultEcore part, where we can specify the MCMT rules (top), the multilevel hierarchy (middle) and the possible specification of behavioural properties that we want to check or enforce during the execution. In [MRS+18b] the so-called supplementary hierarchies are used to define property specification languages like Linear Temporal Logic (LTL) and to specify behavioural properties. We can directly translate these properties to Maude since it implements an LTL model checker. The Transformer: MultEcore ↔ Maude takes care of the automatic transformation. This can be viewed as a bidirectional transformation [Ste07, CFH+09] between the model spaces in MultEcore and Maude:

MultEcore → Maude: once the modeller decides which specific language is going to be simulated, the transformer takes both the models that define the language (the concrete hierarchy branch) and the multilevel model transformation rules, and creates the Maude specification. Such a specification corresponds to a functional Maude file that can be executed directly.

Maude → MultEcore: the states that Maude provides (new versions of the model) are given by means of an XML file. This file is interpreted by the transformer which can directly propagate the new state(s) to the multilevel hierarchy in MultEcore.

The right-hand side of Fig. 1 shows the Maude perspective. Once we have generated the specification with the Transformer, we are able to execute the model using Maude's

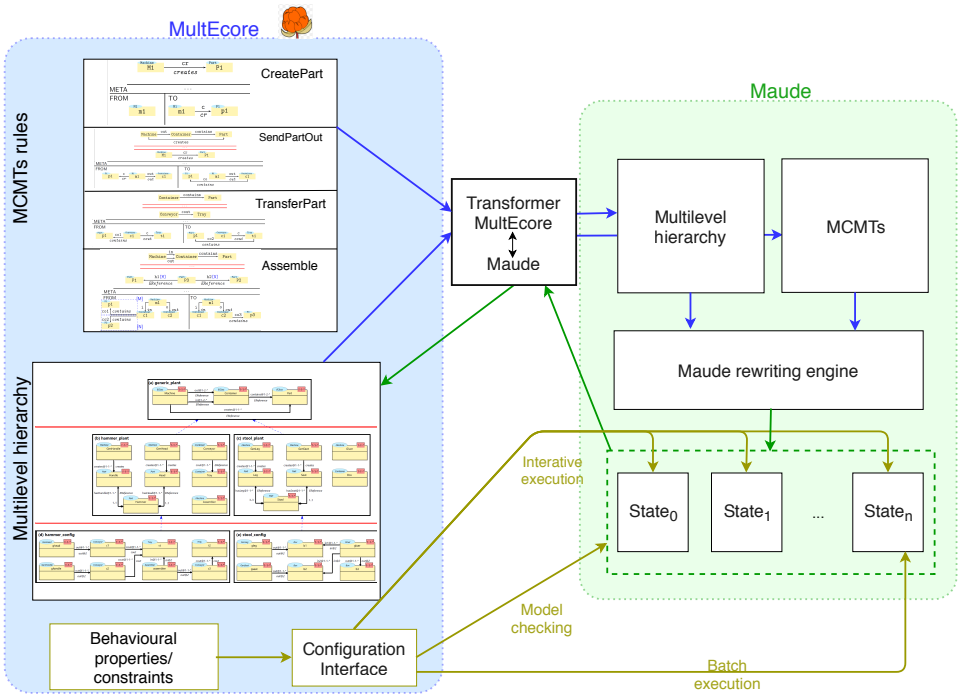

Figure 1 – Infrastructure for the execution of multilevel models

rewriting engine. As Maude allows several kinds of rewriting procedures depending on the strategy chosen, we might want to perform either an interactive execution (i.e., step-by-step, where the modeller can take control of the next states that can be given), or batch execution to directly get a final state.

In our prototype, we fully implement the capability to specify multilevel hierarchies and MCMTs, the bidirectional transformer, and the execution of the specified configurations (see [Dep] to access the infrastructure). This encompasses all the features shown in the figure except for the Configuration Interface which is aimed to offer the modeller a user-friendly interface for controlling aspects related to execution and verification.

3 DSML Structure - Multilevel Modelling

In this section, we discuss how we achieve the definition of the structural dimension of DSMLs by means of MLM. MLM is based on the idea of deep instantiation and eliminating the restriction in the number of times a model element can be instantiated. In this context, MLM techniques match well with the creation of DSMLs, especially when we focus on behavioural languages since behaviour is usually defined at the metamodel level while it is executed at least two levels below; i.e., at the instance level [dLG10, MWR$^+$19].

Usually, when we are defining the structure of a domain-specific language, we mentally "sketch" this as a hierarchical composition. It is therefore natural to have a way to literally translate this mental representation into a model. An example of a multilevel hierarchy (originally from [RDV09]) describing a DSML for Product Line Systems (PLS) is shown in Fig. 2. This hierarchy (which is specified using MultEcore) contains three levels of abstractions (four if we include the reserved level 0 that corresponds to *Ecore* in EMF, and five if we take into account the extension we make in Sect. 4.2). Note that each model in the hierarchy is a directed multi-graph and we establish typing relations in the vertical dimension which are formalised as *graph homomorphisms* [EEPT06]. The complete formalization as well as other MLM examples and an evaluation of MultEcore are depicted in [Mac19]. Further examples of multilevel models with four or more levels can be checked out in [RDLGN15].

The example displays a hierarchical distribution with the generic_plant model at the top (Fig. 2a). In this model, the abstract concepts related to the manufacturing of objects are defined. Machine is aimed for any gear that can create, modify or combine objects, which are represented by the concept Part. Both concepts are linked by the creates relation. A Container can store parts, and this connection is captured by the relation contains. All machines may have containers where they can take parts from or where they can drop the manufactured ones. These two relations are identified with the in and out edges, respectively. The annotations in the rectangles at the right top corners of the nodes, and after the names in the arrows (separated by '@') specify the potencies. *Potency* is used on elements as a means of restricting the levels at which this element may be used to type other elements. In the case of MultEcore, a potency specification includes three values: the first two specify the first and the last levels where one can directly instantiate an element (min and max), and the third value specifies the number of times the element can be indirectly re-instantiated (depth).

The second level contains two models which are defined for two specific environments: one for creating hammers and one for manufacturing stools (hammer_plant in Fig. 2b and stool_plant in Fig. 2c, respectively). One can see that both branches

share similarities. The languages (branches) must belong to the same family in order to make (horizontal) reusability possible. The hammer_plant contains the concepts related to the manufacturing of Hammers which are created by combining one Handle and one Head. This can be seen from the multiplicities 1..1 in the relations hasHandle and hasHead. These two relations have as type EReference (from Ecore [SBPM09]) since no relation is defined between parts in the top level model (generic_plant). This is because the concept of assembling parts is too specific to be located in generic_plant. At this level, we can also find the machines GenHandle and GenHead that create the parts, the Assembler, and the containers Conveyor and Tray that move and store the parts, respectively. It is due to the nature of PLSs that the stool_plant (Fig. 2c) branch in the MLM hierarchy is structured similar to the one in hammer_plant. In this case, we have machines GenLeg and GenSeat to generate Leg and Seat, respectively, and a Gluer that puts together three legs and one seat to make a Stool.

The two models defined at the bottom of the hierarchy, in Fig. 2d and Fig. 2e, represent specific configurations for hammers (hammer_config) and stools (stool_config) productions, respectively. They contain specific instances of the concepts defined in the levels above and they are used to specify concrete product lines configurations, in which parts get transferred from generator machines to machines that combine them.

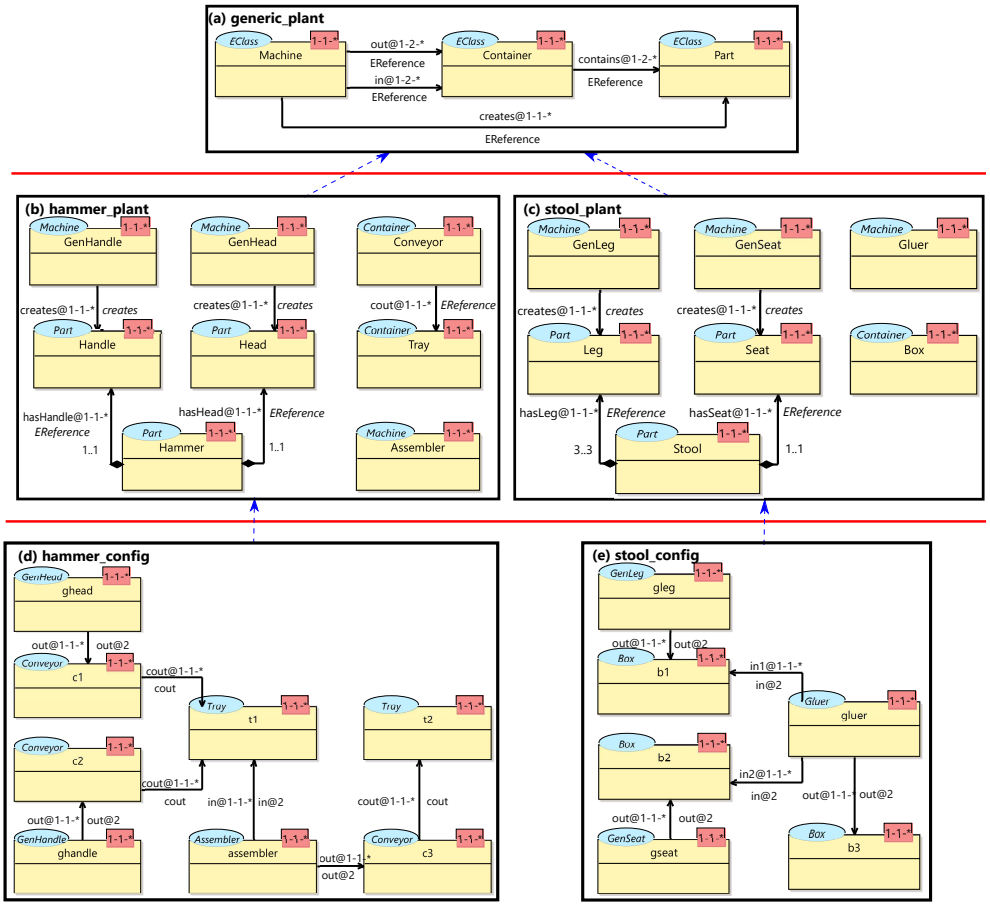

Figure 2 – Full hierarchy for the PLS case study

One could argue that this hierarchy can be managed with traditional two-level approaches using specialization and generalization (i.e., using subclassing and inheritance relations, respectively). The traditional 4-layer architecture of OMG would force us to a design with several concepts in the same model, since this architecture leaves only one level for user models. The top level M3 is reserved for MOF; M2 for metamodels, e.g., UML class diagram or UML object diagram; M1 is designated for user-models; M0 has a "representation" relation to M1, which associates elements of M1 to real world objects, i.e., there is no "instance-of" relationship to the M1 level above. Hence, we would fit the levels generic_plant, hammer_plant and hammer_config into one model at M1 level. Furthermore, the typing relations between model elements in these different levels would have to be maintained manually, i.e., we would need elements like *MachineInstance* and *MachineType*, *ContainerInstance* and *ContainerType*, etc.

MLM provides the flexibility needed to avoid the use of anti-patterns (e.g., type-object pattern is described in [LGC14, LG18]) when fitting several layers of abstractions into one single level. This anti-pattern appears when both the concept and the metaconcept are defined in the same level, leading to convolution. Since the focus and the contribution of this paper is oriented to the flexible definition of the behaviour and the execution/simulation of the models, we do not enter into details of all the concepts related to the definition and construction of MLM hierarchies; we refer to [dLGC15, AK18, Küh18a, Küh18b, MWR+19] for the details.

4 DSML behaviour - MCMTs

Transformation rules can be used to represent actions that may happen in the system. Conventional in-place model transformations (MTs) are rule-based modifications of a source model (specified in the left-hand side of the rule) resulting in a new state of the model (determined by the right-hand side). While the left-hand side takes as input (a part of) a model and it can be understood as the pattern we want to find in our original model, the right-hand side describes the target state of the system we want to acquire in our model. There is a match when what we specify in the left-hand side is found in our source model. The behaviour is the implicit transition from the left-hand side to the right-hand side.

MCMTs have been proposed as a mean to overcome the issues of both the traditional two-level transformation rules and the multilevel model transformations [MWR+19]. While the former lacks the ability to capture generalities, the later is too loose to be precise enough (case distinctions). In this section we show how the behaviour of a multilevel DSML can be described by using MCMTs.

4.1 PLS behaviour definition

The actions illustrated in this section describe a possible behaviour in the PLS environment. These actions detail how to create parts, move them through the different machines and assemble them into new parts. A rule *CreatePart* can be specified as shown in Fig. 3. It represents the process in which a machine creates a part. The META block allows us to locate types in any level of the hierarchy that can be used in FROM and TO blocks.

However, the actual power of the META comes from the fact that it facilitates the definition of an entire multilevel pattern. The rule *CreatePart* is sufficient to generate instances of Head and Handle for the hammer branch of Fig. 2 and

instances of Seat and Leg for the stool branch of Fig. 2. The variable P1 matches to any of the aforementioned parts, both in hammer_plant and stool_plant models, and the variable M1 matches any of the creator machines: GenHead, GenHandle, GenSeat or GenLeg. However, the key feature is that this rule can only match the generators of parts, since we require M1 in the META block to have a creates relation to P1. Then a correct match of the rule comes when an element, coupled together with its type, fits an instance of M1 that has a relation of type creates to an instance of P1. For example, GenHead in Fig. 2b, fits M1, since GenHead has a creates relation to Head. Hence, m1 can be matched to ghead (defined at the left in Fig. 2d) when applying the rule, in order to create a new part (p1), which would be an instance of Head.

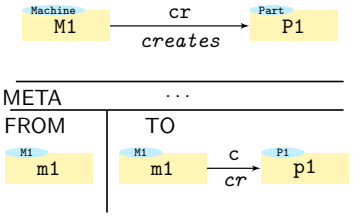

Figure 3 – Rule *CreatePart*: The execution gives a state where a machine has created a part

Compared to the original idea of MCMTs [MWR+19] (the levels specified in a rule had to be consecutive by default) we have removed the strictness in the levels to provide a more flexible definition. There might be several levels in between the blocks FROM/TO and the upper level. This is represented by the three dots in Fig. 3.

Another rule called *SendPartOut* shown in Fig. 4 is the action defined for moving a created part from its generator into the output container. It shows two levels specified in the META block (separated by the upper double line). Similarly as in the *CreatePart* rule, the three dots in between the specified meta levels enhance the flexibility of the rule that can be applied in several cases without modifying it (this will be shown later in this section). Also, it leads to a more natural way of defining that a type is defined at some level above, without the need of saying explicitly in which level. At the top level, we mirror part of generic_plant, defining elements like out and contains, that are used directly as types in the FROM and TO blocks. These elements are defined as constants, meaning that the name of the pattern element must match an element with the same name in the typing chain. The use of constants allows us to be more restrictive when matching, and significantly reduces the amount of matches that we obtain. On the other hand, we allow the type on the variables to be *transitive* (i.e., indirect typing). For instance P1, which has the variable Part for the type, will match any node which indirectly has Part as type, or ultimately will match to Part if no indirect one is found. Fig. 5 displays *TransferPart* rule which moves a part from a Conveyor to a Tray. It models the action where c1 (of type Conveyor), that holds a part p1 and which is connected to t1 with Tray as type (described in the FROM block), moves such a part to t1 (specified in the FROM block).

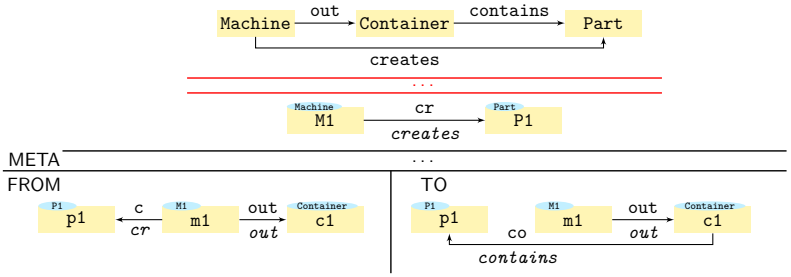

Figure 4 – Rule *SendPartOut*: A part is moved from the creator machine to a container

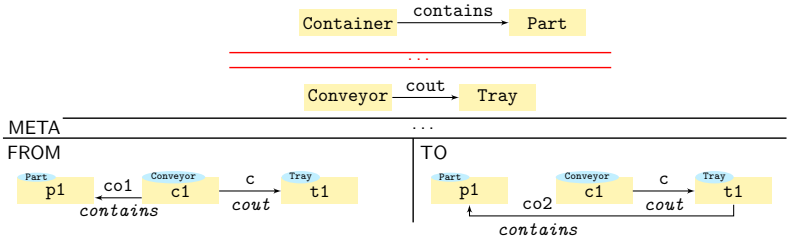

Figure 5 – Rule *TransferPart*: The execution provides a model state where a part is transferred from a conveyor to a tray

The *Assemble* rule creates new products by combining the component parts. It assembles two parts into a different part (see Fig 6). It requires, for the resulting part p3, to consist of, or be built from parts p1 and p2. Having three variables for the different parts, allows us to make an explicit distinction between them even though all of them are instances of Part. Variables [M] and [N] in the intermediate level on the h1 and h2 relations, represent the cardinality that have to be matched in order to apply the rule. A part consisting of other parts might also need a specific number of instances to be built from. In Fig. 2b we can see that a Hammer is composed of 1 Handle and 1 Head (this in fact can be understood as the default case). However, in Fig. 2c, a Stool needs 3 Legs in order to be assembled. As the multiplicity has been explicitly specified in the second META level of the rule, and we have established those same variables for the multiplicities in the FROM block, then the rule will take that into consideration during the matching process. When this process takes action, M and N will be bound to 3 and 1 for stools, respectively. Then, these numbers will be used to check whether that amount of parts exist in the FROM block. For the match to succeed, three legs and one seat (and the respective relations with the container), need to be found. Thus, this is syntactic sugar to represent that in the model it is necessary to explicitly find this number of instances for the match to occur. The way we define and use these multiplicities is inspired by the concept of *cardinality* described in [SCGdL11]. Fig. 7 shows the unfolded version of the *Assemble* rule. As M and N have been bound to 3 and 1, respectively, the pattern shown in the figure needs to be found for a successful match.

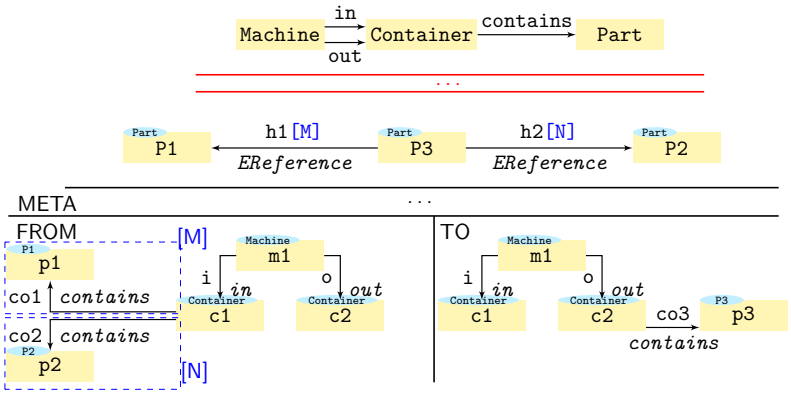

Figure 6 – Rule *Assemble*: The execution gives a state where a machine takes several parts and assemble them in a new one

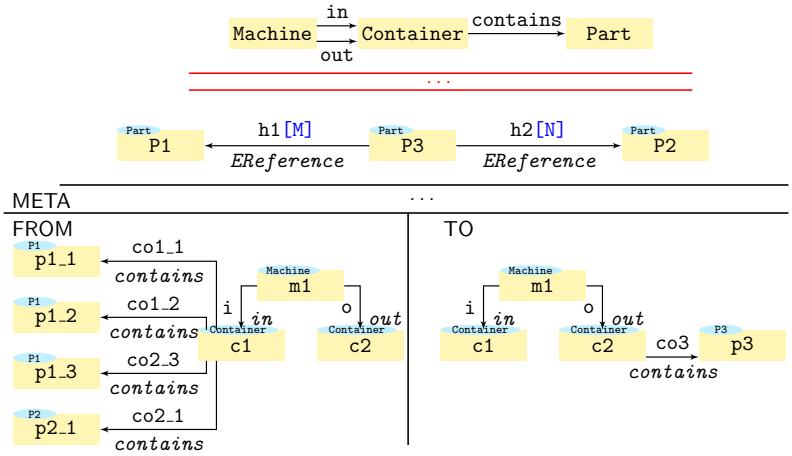

Figure 7 – Unfolded version of *Assemble* rule. The match takes into account the multiplicities specified, and searches for three p1 and one p2

4.2 Horizontal and vertical flexibility

In the previous section we have defined the model transformation rules that provide the behaviour to the multilevel hierarchy. Horizontal flexibility is indirectly inferred since these rules can be directly applied to both branches shown in Fig. 2. For example, *CreatePart* rule can be applied to create either a Head or a Handle (for hammer branch) or to create a Leg or a Seat (for stool branch).

In this section, we demonstrate how MCMT rules are still applicable when modifying an existing multilevel hierarchy (vertical flexibility) and how we can make restrictions in the rules to confine the typing flexibility.

Let us suppose that ACME factories have some specific type of hammers that are created by a handle and a green head. This can be introduced as a new level in between Fig. 2b and Fig. 2d, that captures the ability to create green heads, called special_head_config. This new level is depicted in Fig. 8. The two nodes, SpecialGenHead and GreenHead and the edge creates are now instances of GenHead, GreenHead and creates, respectively, which are defined in the level hammer_plant (Fig. 2b).

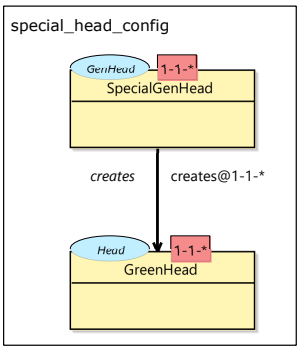

Figure 8 – New specified level for creating green heads

We are now able both to define a generator for regular heads and also a generator for green heads, in the level shown in Fig. 2d. As this depends on the concrete scenario, we might construct different configurations which can include any combination of the two generator of heads aforementioned, and the rules should be agnostic to those possibilities. Fig. 9 shows the two possible matches depending on the machine we define at the instantiation level (i.e., at the lowest level). At the left side of the dashed double vertical line we can see the *CreatePart* rule, already shown in Fig. 3.

At the right side we show a hierarchy consisting of three levels (divided by horizontal lines). These levels comprise those elements of the PLS hierarchy involved in the generation of heads (i.e., a *generator* that *creates* a *head*). They represent

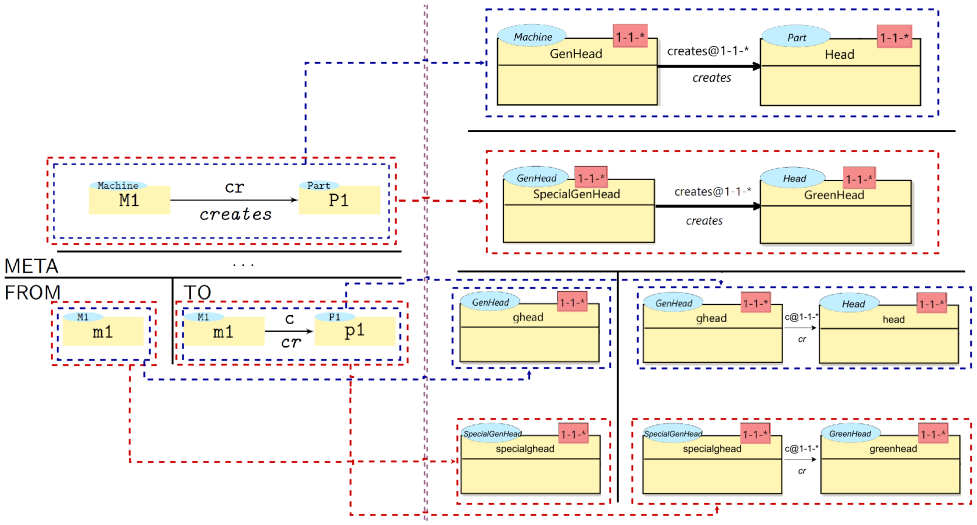

Figure 9 – Vertical reusability of rule *CreatePart*

hammer_plant (Fig. 2b), *special_head_config* (Fig. 8) and *hammer_config* (Fig. 2d) levels, respectively. Note that the lowest level shown in this hierarchy is divided by a vertical black line, which represents the same logic as the FROM/TO pattern. This level is composed by two instances, one represents the match for the creation of a regular head (at the top) and the other corresponds to the match of a specified generator of green heads (at the bottom).

The dashed blue lines represent the match of the rule in case we define our configuration as using regular head generator (ghead), while the dashed red lines represent the match of the rule for a scenario where we have a green head generator (specialghead). Moreover, the right hand side of the hierarchy at the instantiation level (bottom-right side of Fig. 9) shows the state where the *CreatePart* rule has been fired. As one can observe, the rule has not been modified at all, but the flexibility provided allows both matchings depending on the scenario specified.

The default flexibility opens for several possible matchings. For instance, a normal head could be created by a special head generator. Another possibility is, in the *Assemble* rule, that a hammer can be manufactured from a green head and a normal handle. Since considering these matches as valid is up to the modeller, we provide functionality for allowing/disallowing them. We can restrict the *CreatePart* rule using a matching strategy where the nearest type is selected (specialization priority) and still leave open the matches for *Assemble*.

One might consider the need of restricting the indirect typing which is allowed by-default since this flexible assumption (*the type can be found at any number of jumps of any length*) might not be desired in all situations. To disallow that, we can use $t@n$ $(n \mid n \in \mathbb{N})$ over a type t. First, this disables the indirect typing (so we must find the type in just one jump upwards) and second, it forces the type to be at n levels above the one where the match for t has been found.

5 Formal specification and execution with Maude

As explained in Section 1, the MultEcore tool allows us to define multilevel hierarchies and MCMTs to describe their behavior. MultEcore relies on Maude for the simulation and formal analysis of the specified MLM systems.

Maude [CDE+07] is a specification language based on rewriting logic [Mes92], a logic of change that can naturally deal with states and non deterministic concurrent computations. A rewrite logic theory is a tuple $(\Sigma;E;R)$, where Σ is called signature and specifies the type structure (sorts, subsorts, etc.) and E is the collection of equations and memberships declared in the functional module. Therefore, $(\Sigma;E)$ is an equational theory that specifies the system states as elements of the initial algebra $\mathcal{T}_{(\Sigma;\mathrm{E})}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system. Rewrite specifications thus described are executable, since they satisfy some restrictions such as termination and confluence of the equational subspecfication and coherence of equations and rules. Indeed, Maude provides support for rewriting modulo associativity, commutativity and identity, which perfectly captures the evolution of models made up of objects linked by references as in graph grammar. In summary, Maude provides, among others, the next useful features [CDE+02]:

Formal specification. The Maude specification of multilevel hierarchies and MCMTs represents a formal semantics in rewriting logic. Since these specifications are executable, they can be used for simulating/executing our models. The automatic bidirectional transformation MultEcore $\leftrightarrow$ Maude allows the execution of MLM models from the MultEcore tool. Indeed, Maude's flexibility and customization capabilities have allowed us to represent MLM models and MCMTs in Maude using a syntax very similar to the MultEcore syntax. This has led to a straightforward transformation between MultEcore and Maude.

Execution of the specification. The Maude specification obtained from MLM hierarchies and corresponding MCMTs using the above transformation are executable, and therefore can be used to simulate them in Maude. The versatile rewriting engine provides a lot of functionalities to customize the way we go trough the execution steps. As we will see below, we can simulate our systems letting Maude choose the path to follow, or we can specify a concrete path by means of execution strategies.

Formal environment. Once the rewriting logic specification of the MLM hierarchies and their MCMTs is available in Maude, we can use the formal tools in its formal environment to analyze the systems thus described. For example, we can check properties as confluence or termination of our specifications, but also perform reachability analysis, model checking or theorem proving on them.

5.1 Multilevel hierarchies in Maude

In the Maude language, object-oriented systems can be specified by object-oriented modules in which classes and subclasses are declared, with the usual support for inheritance, dynamic binding, etc. A class is declared with syntax class $C \mid a_1\colon S_1, \ldots, a_n\colon S_n$, where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. The objects of a class C are record-like structures of the form $< O : C \mid a_1\colon v_1, \ldots, a_n\colon v_n >$, where O is the identifier of the object and v_i are the current values of its attributes.

```
1  class Model | name : Name, om : Name, elts : Configuration, rels : Configuration .
2  class Element | name : Oid, type : Oid .
3  class Node | .
4  class Relation | source : Oid, target : Oid, min-mult : Nat, max-mult : Nat* .
5  subclasses Node Relation < Element .
```

Figure 10 – Maude structure of a multilevel hierarchy

In a concurrent object-oriented system, the concurrent state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting using rules that describe the effects of the communication events of objects and messages. The system presented in this paper evolves as the result of applying the rewrite rules on collections of objects.

A multilevel hierarchy is represented in Maude as a structure of sort System of the form $MLM\{model_1\ model_2\ \ldots model_n\}$, where MLM is the name of the multilevel hierarchy and each $model_i$ is an object of class Model that represents a model in the hierarchy. Note that when the transformation from MultEcore to Maude is to be performed, the modeller have to decide the branch that is it going to be executed. For instance, for the PLS hierarchy used in this paper, the modeller would decide between hammer or stool branch. Fig. 10 illustrates the specification of a multilevel hierarchy in Maude. A model is represented as an object of class Model, which has attributes representing its name, its ontological metamodel om, and the nodes (elts) and relations (rels) that are part of it (line 1). As mentioned in the previous paragraph, Configuration is a predefined sort in Maude implemented to deal with object-based systems. Instances of classes Node (line 3) and Relation (line 4) represent, respectively, nodes and relations. Both are subclasses (line 5) of a class Element (line 2) of elements with a name and a type. In addition to the attributes inherited from Element, class Relation has attributes for the source and target of a relation, and its multiplicity range (min-mult, max-mult).

The sort Name allows us to define how our objects are going to be identified. For instance, the identifier of a model is represented as $level(x)$, for x either 0 or a natural number. Level 0 is always reserved for *Ecore* and n the lowest level in the hierarchy. Identifiers are required to be unique. The transformation assigns these names automatically when generated. As we will see in the next section, objects generated in transformation rules are also given unique fresh names.

Given these declarations, Fig. 11 illustrates how models are represented. Specifically, it shows the Maude term that represents the generic_plant level (corresponding to Fig. 2a). Note that this is just one of the models in the MLM hierarchy. In this case it has assigned level(1) (Line 1). Then we have the name of the model ("generic-plant", Line 2) and its metamodel at the level right above represented by om : "Ecore" (Line 3). It contains two sets, elts (Line 4) and rels (Line 8), for capturing the nodes and the relations, respectively. For instance, the relation specified in Line 11, with identifier oid(1,5), represents the *in* relation between a machine and a container: its name is id(1,"in"), its type is id(0, "EReference"), and it links two nodes, id(1, "Machine") as source (in Line 5) and id(1, "Container") as target (in Line 6). As expected, all the information available in MultEcore is encoded in the Maude representation.

5.2 The MCMT rules in Maude

In Maude, a distributed system is axiomatized by an equational theory describing its states as an algebraic data type and a collection of conditional rewrite rules specifying its *behaviour*. Rewrite rules are written crl $[l] : t \Rightarrow t'$ if C, with l the rule label, t and t' terms, and C a guard or condition. Rules describe the local, concurrent transitions that are possible in the system, i.e., when a part of the system state fits the pattern t, then it can be replaced by the corresponding instantiation of t'. The guard C acts as a blocking precondition: a conditional rule can only be fired if its condition is satisfied. Rules may be given without label or condition.

We can directly translate MCMT rules to conditional rewrite rules in Maude. We illustrate this representation of rules with the *CreatePart* rule in Fig. 3. Fig. 12 shows its Maude counterpart. The left-hand side of the rule (Lines 2-17) encodes the META and FROM blocks of the rule. In this case, in the META section there is one model level(L), and in the FROM one model level(J). Notice that we are not specifying a concrete level, but we use variables that will be bound when the rewriting engine matches the rule to our MLM concrete hierarchy. It is in the conditions where we can constraint the behaviour of the rule by defining predicates or conditional expressions, such as L < J. The counter specified in Line 16 is an auxiliary object that keeps a counter so that we can create fresh new identifiers for the elements created in the right-hand side. The rest of the left-hand side is fairly straightforward as it can directly be inferred from the *CreatePart* rule displayed in Fig. 3). Atts..., Elts..., O..., etc. are just variables we define to capture those attributes we do not explicitly specify.

The right-hand side of the rule (Lines 18-29) shows how the objects in the left-hand side will be modified when the rule is applied; the ellipses are only to save space. Model level(L) (the META) is left unmodified. We display in detail the level(J) model, which corresponds to the TO block shown in Fig. 3. As we can see in Lines 22 and 24-25, there are two new elements. The first one corresponds to the new part, which will have identifier oid(J, s N), name id(J, s s s N), and type P1 (note the reference to the correspondent object in model level(L)). Notice that new identifiers and names are generated by using the counter object. The s operator is a Maude predefined operator that calculates the successor of a number. For instance, if J and N get bound to 3 and 100, we would get as results oid(3, 101) and id(3,103), respectively. A new relation is also created, with source m1 (Line 12) and target the new part id(J, s s s N).

In addition to the condition on models level(L) and level(J), other conditions are also

```
1  < level(1) : Model |
2      name : "generic-plant",
3      om   : "Ecore",
4      elts : (
5        < oid(1,1): Node | name : id(1, "Machine"), type : id(0, "EClass") >
6        < oid(1,2): Node | name : id(1, "Container"),type : id(0, "EClass") >
7        < oid(1,3): Node | name : id(1, "Part"), type : id(0, "EClass") >),
8      rels : (
9        < oid(1,4): Relation | name : id(1, "out"), type : id(0, "EReference"),
10           source : id(1, "Machine"), target : id(1, "Container") >
11       < oid(1,5): Relation | name : id(1, "in"), type : id(0, "EReference"),
12           source : id(1, "Machine"), target : id(1, "Container") >
13       < oid(1,6): Relation | name : id(1,"contains"), type : id(0,"EReference"),
14           source : id(1, "Container"), target : id(1, "Part") >
15       < oid(1,7): Relation | name : id(1,"creates"), type : id(0,"EReference"),
16           source : id(1, "Machine"), target : id(1, "Part") >) >
```

Figure 11 – Maude specification for generic_plant model

given as a conjunction of predicates. In this case, * references of variables are handled by the predicate *. This predicate is necessary to provide a type with the *transitive* dimension mentioned in Sect. 4.1. We call it * to be consistent with the original idea (degree of genericness $*t$) presented in [MRS+18b]. Multiplicities, potencies, and other facilities in the rules are handled similarly.

5.3 Execution and results

Given MLM hierarchies and MCMT rules specified in Maude as shown in the previous section, we have several options for executing it. Given an initial ground MLM hierarchy instantiation from which to start the execution, the Maude rewrite commands can attempt the consecutive application of the rules in our specification. Maude provides two different rewriting commands, for which we can specify a maximum number of rewriting steps to take, implementing two different strategies: *rewrite* follows a top-down rule-fair strategy and *frewrite* follows a depth-first position-fair strategy.

In addition, Maude also provides commands for the controlled execution of our rules. Maude facilitates a rich strategy language with which we can specify our own strategies. For example, we can perform a batch execution, just by specifying step by step, which rules are to be applied, and, if desired, the objects on which it should happen, by providing a partial substitution for the instantiation.

Let us show a very simple example of the use of the *srewrite* command (abbreviated *srew*), which allows us to apply a concrete strategy to a given term (our initial state will

```
1    crl [CreatePart] :
2      { < level(L) : Model |
3            name : M,
4            elts : (< O01 : Node  | name : M1, type : *Machine, A01 >
5                    < O02 : Node  | name : P1, type : *Part, A02 >
6                    Elts),
7            rels : (< O03 : Relation | name : cr, type : *creates, source : M1, target : P1,A03>
8                    Rels),
9            Atts >
10       < level(J) : Model |
11           name : M',
12           elts : (< O04 : Node | name : m1, type : *M1, A04 >
13                   Elts'),
14           rels : Rels',
15           Atts' >
16       < counter : Counter | value : N >
17       Conf }
18    => { < level(L) : Model | ... >
19       < level(J) : Model |
20           name : M',
21           elts : (< O04 : Node | name : m1, type : *M1, A04 >
22                   < oid(J, s N) : Node | name : id(J, s s s N), type : P1 >
23                   Elts'),
24           rels : (< oid(J, N) : Relation | name : id(J, s s N), type : cr,
25                     source : m1, target : id(J, s s s N), min-mult : 1, max-mult : 1 >
26                   Rels'),
27           Atts' >
28       < counter : Counter | value : s s s s N >
29       Conf }
30    if L < J
31    /\ *(*M1, level(sd(J,1)), M1, ...)
32    /\ *(*Machine, level(sd(L,1)), id(1, "Machine"), ...)
33    /\ *(*Part, level(sd(L,1)), id(1, "Part"), ...)
34    /\ *(*creates, level(sd(L,1)), id(1, "creates"), ...) .
```

Figure 12 – Maude representation of the *CreatePart* rule (note the ellipses)

be the Maude representation of either hammer_config (Fig. 2d) or stool_config (Fig. 2e)) where we specify the rules (and optionally some constraints within them) ordering. Let us assume that we want to make a complete iteration over the hammer_config model (for a smoother explanation, in this scenario we do not consider having the specialghead as instance of SpecialGenHead displayed in Fig. 8 and described in Sect. 4). We would need then to create a head, a handle, and eventually we would get assembled a new hammer. Fig. 13 shows the defined strategy we can execute to test if such a final state is reached.

```
1  srew PLS using CreatePart ;
2                 CreatePart ;
3                 SendPartOut ;
4                 SendPartOut ;
5                 TransferPart ;
6                 TransferPart ;
7                 Assemble ;
8                 TransferPart .
```

Figure 13 – Strategy for manufacturing a Hammer in hammer_config configuration

PLS in Line 1 corresponds to the initial term (the complete hierarchy). As the strategy is written, each application of the CreatePart rule (Lines 1 and 2) can create either a Handle or a Head. This is not a problem as the rewriting engine provides all the solution, and then it discards the non valid ones when applying the Assemble rule (a solution right before the execution of Assemble might have produced 2 Handles). However, we can constrain the applications of the CreatePart rule by explicitly providing a partial substitution:

```
CreatePart[P1 <- id(2, "Handle")] ;
CreatePart[P1 <- id(2, "Head")] ;
```

With this, we are binding P1 to generate first a Handle, then a Head. In both cases, we would end up having the same solution.

The rest of the rules are applied sequentially. Taking the model in Fig. 2d as reference, we would create a handle and a head, and move them to the conveyors c1 and c2, respectively, using the rule *SendPartOut*. Then we move both parts to the tray t1 with the rule *TransferPart* and the parts are assembled into a hammer using the rule *Assemble*. Finally, the hammer is moved from the conveyor c3 to the tray t2 using again the rule *TransferPart*. Once we get a solution model by running the rules, Maude generates an XML file which is transformed back into MultEcore. Fig. 14 shows how the solution would look in the graphical view of MultEcore.

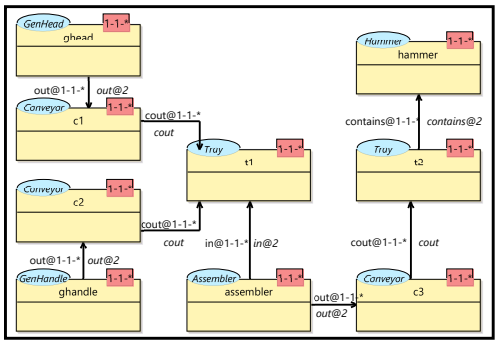

Figure 14 – hammer_config with a Hammer

6 Related work

There exist several tools and technologies that support dynamic execution of models through model transformation in a graphical manner. A Tool for Multi-Paradigm Modeling (AToMPM) [SVM+13], the Foundational UML (fUML) [Sub11] and the Executable Meta-Object Facility (xMOF) [MLWK13], are some examples that grant such capability. AToMPM is an open-source framework for designing DSML environ-

ments, performing model transformations, manipulating and managing models which runs entirely over the web. The fUML is an executable subset of UML that can be used to define, in an operational style, the structural and behavioural semantics of systems. However, due to its exclusive focus on UML, it cannot be applied to arbitrary domain-specific languages. The Action Language for Foundational UML (ALF) [Sei14], which is built on top of fUML, provides functionality for executing UML models in a textual way. They both are intended to work together, resulting in a more complete framework that provides both graphic and programmatic (when a high degree of details is required) facets. The xMOF is a metamodelling language that integrates Ecore with the behavioural part of fUML. It is aimed at developing executable DSMLs that can be simulated using the fUML virtual machine. In [BEK+06], the authors present an approach for the definition of in-place transformations in EMF. All the approaches mentioned above are based on traditional two-level approaches which disallow the multilevel capabilities presented in this paper.

ConceptBase [JGJ+95] is a tool that implements the object model of a Datalog-based variant of Telos [MBJK90]. It supports subtyping chains and unrestricted class-instance relationships. However, it does not make a clear organization of elements in hierarchical models. Furthermore, it does not support key features of MLM as flexible depth (supported by our approach via *Potency* concept). The MOMENT-QVT tool [BCR06] is a model transformation engine that provides partial support for the QVT relations language [RVA06]. QVT (Query/View/Transformation) is a standard set of languages for model transformation defined by the OMG. In [AGT12], authors present an approach to transform from a multilevel setting to a two-level configuration (and the other way around) using the ATL Transformation Language (ATL) [JABK08], which is not designed to work within a multilevel context. However, our approach makes it possible to directly define the behaviour of our multilevel hierarchy (by using MCMTs). Since we can directly translate and use MLM hierarchies in Maude, a transformation to a two-level setting to be able to rely on a model transformation engine (like ATL or QVT [Kur07]) is not necessary.

In [RGdLV08], the authors show how Maude is used to represent a subset of the PLS example used in this paper. The subset corresponds to the left-hand branch of the multilevel hierarchy shown in Fig. 2. In their work, they encode both the PLS metamodel and an instance of it, to later be able to simulate it and perform formal analysis and model checking. Changes in either the metamodel or the model would need to be done manually in the Maude implementation. Our approach hides the Maude implementation so the user can make modifications directly in the graphical editor which are in turn translated automatically to Maude.

7 Conclusions and future work

In this paper, we have described how flexible and reusable model transformations (by means of the MCMTs) can be applied in the context of MLM. In addition to a theoretical foundation, we have developed a prototype of an infrastructure to connect our MLM tool MultEcore with Maude in order to execute/simulate the constructed models. We have showcased the flexibility and reusability of the MCMT rules, first, in the vertical aspect by adding an extra level into an MLM hierarchy, and second, in the horizontal aspect by using the same rules for two branches of the hierarchy. Furthermore, two important new features have been successfully applied. First, the default restriction forcing the levels in the rules to be consecutive has been lifted,

providing vertical flexibility. Second, multiplicities are now supported in the MCMTs, enriching the syntax and enhancing the reusability of the rules.

We see several directions for future work. The infrastructure that connects MultEcore with Maude has been constructed as a proof of concept, and we are working on considerable extensions. To generalise from the examples in this paper, we will design an experiment in which we pick several mainstream behavioural models, refactor them to MLM hierarchies using [LG18], adapt them to MultEcore using the rearchitecter tool presented in [MRS18a], and then execute them using the presented infrastructure.

We have developed a first version of the multiplicities (cardinalities) in MCMTs. However, we intend to further extend this feature for more complex cases with potential nested definitions. We plan also to provide MultEcore functionalities to control the Maude execution directly from the editor. Moreover, we want to give the user the control to make executions customizable so that step-by-step or batch simulations might be performed. Another task is to provide the MultEcore-Maude transformation engine with more comprehension so that it becomes more fault tolerant. We currently offer the user the possibility to define both the multilevel hierarchy and the behavioural rules. We also want to work on improving the part of the infrastructure for the definition of behavioural properties to later verify them with Maude, in a user-friendly manner. Furthermore, we are currently working on better ways to specify rule orchestration and prioritization to improve the definition and application of strategies in a more generic, reusable and user-friendly way.

References

[AGM15] Colin Atkinson, Ralph Gerbig, and Noah Metzger. On the execution of deep models. In *EXE@ MoDELS*, pages 28–33, 2015. URL: `http://ceur-ws.org/Vol-1560/paper5.pdf`.

[AGT12] Colin Atkinson, Ralph Gerbig, and Christian Tunjic. Towards multi-level aware model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 208–223. Springer, 2012.

[AK08] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[AK17] Colin Atkinson and Thomas Kühne. On evaluating multi-level modeling. In *MoDELS*, 2017.

[AK18] Colin Atkinson and Thomas Kühne. Deep instantiation. In *Encyclopedia of Database Systems, Second Edition*. Elsevier, 2018. `doi:10.1007/978-1-4614-8265-9\_80608`.

[AKdL18] Colin Atkinson, Thomas Kühne, and Juan de Lara. Editorial to the theme issue on multi-level modeling. *Softw. Syst. Model.*, 17(1):163–165, February 2018. `doi:10.1007/s10270-016-0565-6`.

[BCR06] Artur Boronat, José Á. Carsí, and Isidro Ramos. Algebraic specification of a model transformation engine. In *International Conference on Fundamental Approaches to Software Engineering*, pages 262–277. Springer, 2006.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place

transformations in the eclipse modeling framework. In *MoDELS*, pages 425–439. Springer, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[CDE+02] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and José F Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[CFH+09] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *International Conference on Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

[Dep] HVL Computer Science Department. MultEcore Maude Website. URL: `https://ict.hvl.no/multecore-maude/`.

[dLG10] Juan de Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS*, pages 16–30, 2010. `doi:10.1007/978-3-642-16145-2\_2`.

[dLGC15] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software and System Modeling*, 14(1):429–459, 2015. `doi:10.1007/s10270-013-0367-z`.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom 3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. URL: `https://doi.org/10.1007/3-540-31188-2`, `doi:10.1007/3-540-31188-2`.

[EMOMV07] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11):3–25, 2007.

[GM13] Joseph A Goguen and Grant Malcolm. *Software Engineering with OBJ: algebraic specification in action*, volume 2. Springer Science & Business Media, 2013.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[JGJ+95] Matthias Jarke, Rainer Gallersdörfer, Manfred A Jeusfeld, Martin Staudt, and Stefan Eherer. Conceptbase—a deductive object base for

meta data management. *Journal of Intelligent Information Systems*, 4(2):167–192, 1995.

[Küh18a] Thomas Kühne. Exploring potency. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 2–12, 2018. doi:10.1145/3239372.3239411.

[Küh18b] Thomas Kühne. A story of levels. In *Proceedings of MULTI Workshop: co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems*, pages 673–682, 2018. URL: http://ceur-ws.org/Vol-2245/multi_paper_5.pdf.

[Kur07] Ivan Kurtev. State of the art of QVT: A model transformation language standard. In *International Symp. on Applications of Graph Transformations with Industrial Relevance*, pages 377–393. Springer, 2007.

[LG18] Juan de Lara and Esther Guerra. Refactoring multi-level models. *ACM Trans. Softw. Eng. Methodol.*, 27(4):17:1–17:56, November 2018. doi:10.1145/3280985.

[LGC14] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 24(2):12, 2014.

[Mac19] Fernando Macías. *Multilevel modelling and domain-specific languages*. PhD dissertation, University of Oslo, Norway, 2019.

[MBJK90] John Mylopoulos, Alex Borgida, Matthias Jarke, and Manolis Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems (TOIS)*, 8(4):325–362, 1990.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MGS+13] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernández, Bjørn Nordmoen, and Mathias Fritzsche. Where does model-driven engineering help? Experiences from three industrial cases. *Software & Systems Modeling*, 12(3):619–639, 2013.

[MLWK13] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs based on fUML. In *SLE*, pages 56–75. Springer, 2013.

[MRS16] Fernando Macías, Adrian Rutle, and Volker Stolz. Multecore: Combining the best of fixed-level and multilevel metamodelling. In *MULTI@ MoDELS*, pages 66–75, 2016.

[MRS18a] Fernando Macías, Adrian Rutle, and Volker Stolz. A tool for the convergence of multilevel modelling approaches. In *MULTI@ MoDELS*, 2018.

[MRS+18b] Fernando Macías, Adrian Rutle, Volker Stolz, Roberto Rodriguez-Echeverria, and Uwe Wolter. An approach to flexible multilevel modelling. *Enterprise Modelling and Information Systems Architectures*, 13:10:1–10:35, 2018.

[MWR+19] Fernando Macías, Uwe Wolter, Adrian Rutle, Francisco Durán, and Roberto Rodriguez-Echeverria. Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. *Journal of Logical and Algebraic Methods in Programming*, 2019. doi:10.1016/j.jlamp.2018.12.005.

[RDLGN15] Alessandro Rossini, Juan De Lara, Esther Guerra, and Nikolay Nikolov. A comparison of two-level and multi-level modelling for cloud-based applications. In *ECMFA*, pages 18–32. Springer, 2015.

[RDV09] Jose E. Rivera, Francisco Durán, and Antonio Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 51–55. IEEE, 2009.

[Ren03] Arend Rensink. The groove simulator: A tool for state space generation. In *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pages 479–485. Springer, 2003.

[RGdLV08] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In *International Conference on Software Language Engineering*, pages 54–73. Springer, 2008.

[RVA06] Sreedhar Reddy, R Venkatesh, and Zahid Ansari. A relational approach to model transformation using qvt relations. *TATA Research Development and Design Centre*, pages 1–15, 2006.

[SBMP08] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[SCGdL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: Write once, reuse everywhere. In *ICMT*, pages 62–77, 2011. doi:10.1007/978-3-642-21732-6_5.

[Sei14] Ed Seidewitz. UML with meaning: executable modeling in foundational UML and the Alf action language. In *HILT*, pages 61–68. ACM, 2014.

[Ste07] Perdita Stevens. A landscape of bidirectional model transformations. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 408–424. Springer, 2007.

[Sub11] OMG Semantics Of A Foundational Subset. For executable UML models (fUML), version 1.0, 2011.

[SVM+13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A web-based modeling environment. In *MoDELS*, pages 21–25, 2013.

[UML] UML. http://www.uml.org/.

[WHR14] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2014.

About the authors

Alejandro Rodríguez is a PhD student at the Western Norway University of Applied Sciences. He is currently researching in Model-driven software engineering, multilevel modelling and coloured Petri net fields. He is part of the Software Engineering, Sensor Networks and Engineering Computing department. Contact him at `arte@hvl.no`

Francisco Durán is Full Professor at the Department of Computer Science of the University of Málaga, Spain. He received his Ph.D. degree in Computer Science from the University of Málaga in 1999, after several years as an International Fellow at SRI International, CA. He is one of the developers of the Maude system, and his research interests deal with the application of formal methods to software engineering, including topics such as cloud systems, model-driven engineering, component-based software development, open distributed programming, reflection and meta-programming, and software composition.

Adrian Rutle is Associate Professor at the Western Norway University of Applied Sciences, Norway. His research focuses on the application of theoretical results from the field of model-driven software engineering. His work has recently focused on modelling and simulation for smart robotics, MLM, patient workflows and their verification. His main expertise is the development of formal modelling frameworks for domain-specific modelling languages, graph-based logic for reasoning about static and dynamic properties of models, and the use of model transformations for the definition of semantics of modelling languages.

Lars Michael Kristensen received the PhD in computer science from University of Aarhus, and is currently professor in software engineering at Western Norway University of Applied Sciences. He has published more than 70 papers in strictly referred journal and conferences, is member of the Editorial Board of the TopNoC Springer journal, and is a member of the steering committee for the International Petri Nets conference. He is co-author of the most recent textbook on Coloured Petri Net and CPN Tools which is one of the most widely used software tools for modelling and validation of concurrent systems.

Acknowledgments Francisco Durán was partly funded by the project PGC2018-094905-B-I00 (Spanish MINECO/FEDER), and by Univ. Málaga, Andalucía Tech.