Noname manuscript No.
(will be inserted by the editor)

Execution and Analysis of MultEcore Multilevel Modelling Languages using Maude

Alejandro Rodríguez[1] · Francisco Durán[2] · Lars Michael Kristensen[1]

Received: date / Accepted: date

Abstract Multilevel Modelling (MLM) approaches make it possible for designers and modellers to work with an unlimited number of abstraction levels when specifying domain-specific modelling languages (DSMLs). Even though there exists plenty of work in the literature to support MLM solutions from a structural point of view, there is no consensus on how to specify the behaviour of such models. In this paper, we present a functional infrastructure that allows modellers to define the structure and the operational semantics of multilevel modelling hierarchies that can be later simulated and analysed. Using the MultEcore tool, one can design and distribute the models that compose the language family in a multilevel hierarchy, and specify their behaviour by means of multilevel transformation, so-called Multilevel Coupled Model Transformations (MCMTs). This work extends these MCMTs to describe the behaviour of MLM systems with basic support for attribute manipulation, rule conditions, and possibly nested boxes to handle submodel collections. We give a rewrite logic semantics to MLM, on which we have based our automated transformation from MultEcore to the rewriting logic language Maude. Then, we rely on Maude to simulate/execute MultEcore models and to exploit different analysis techniques supported by Maude, like reachability analysis, bounded and unbounded model checking of invariants and LTL formulas on systems with both finite and infinite reachable state spaces using equational abstraction. We illustrate our developed techniques on a DSML family for Petri nets.

Keywords Multilevel Modelling · Domain-specific modelling languages · Model transformations · Verification · Rewriting logic · Maude

✉ Alejandro Rodríguez
arte@hvl.no

Francisco Durán
duran@lcc.uma.es

Lars Michael Kristensen
Lars.Michael.Kristensen@hvl.no

[1] Western Norway Univ. of Applied Sciences, Bergen, Norway
[2] ITIS Software, University of Málaga, Málaga, Spain

1 Introduction

Multilevel Modelling (MLM) is a notable research area where models and their specifications can be organised into several levels of abstraction [4]. Indeed, the MLM community has shown that MLM is a favourable approach in domains such as process modelling and software architecture [6,8]. Although there exist diverse approaches for MLM (see [32,2,61,64] for some of them), they all share a common idea: lift the restriction on not limiting the number of levels that designers can use to specify modelling languages.

This restriction is present in traditional Model-Driven Software Engineering (MDSE) approaches which are based on the Object Management Group (OMG) [44] 4-layer architecture such as the Unified Modelling Language (UML) [63] and the Eclipse Modelling Framework (EMF) [59,45]. Like in traditional MDSE approaches, MLM uses abstractions and modelling techniques to tackle the continually increasing complexity of software by considering models as primary artefacts in each phase of the software engineering life-cycle [11]. Using MLM, modellers are no longer forced to fit their modelling language specifications within two levels of abstraction: one for (meta)models

and one for their instances. This might be too restrictive for certain situations where the language is large and/or complex, and even more when defining behavioural domain-specific modelling languages (DSMLs). DSMLs that are, for instance, variations on general purpose languages, i.e., to specify different refinements aimed at specific domains, would require further concretisations of the metamodels. Moreover, these limitations may lead to complications like model convolution, accidental complexity, and mixing concepts belonging to different domains (see, e.g., [34,6,7] for discussions on these issues).

One of the most prominent applications of MDSE is the construction of DSMLs [45]. These are modelling languages that are tailored to a concrete application area [28] which bridges the gap between software engineers and domain experts. DSMLs are usually built on top of a more abstract modelling language, which requires well-defined infrastructures to handle the separation of different abstraction levels. Furthermore, MLM techniques are excellent for the creation of DSMLs, especially when focusing on behavioural languages, since behaviour is usually defined at the metamodel level while it is executed (at least) two levels below at the instance level [33,3,39]. The reason is that behaviour is reflected in the running instances of the models which in turn conform to their metamodel.

The approach for MLM proposed by the tool MultEcore [38,56], formally specified in [37], rests on the premise that one must be able to specify models (distributed along tree-like hierarchies) which are both generic and precise [39]. Even though various approaches have been proposed for the definition and simulation of behavioural models based on reusable model transformations (e.g., [49,35,51]), these rely on traditional two-level modelling hierarchies. Furthermore, modelling the behaviour through multilevel model transformations [3] and performing execution or analysis in MLM has not been widely explored yet.

Having a hierarchical organisation of the models that are in fact separate artefacts which altogether precisely capture the desired system facilitates future extensions and modifications. This applies not only in the existing levels, but also for adding or removing models to the existing multilevel hierarchy. Therefore, it can help to prevent pollution of models where specialisation of concepts would have to be done in the same model (even if they naturally fit in different levels of abstraction). Furthermore, this enhances modularisation and facilitates extendibility [57].

To cope with execution/simulation of models within the MLM context, Multilevel Coupled Model Transformations (MCMTs) were formally introduced in [39] as a multilevel transformation language that bridges the gap for the execution of multilevel modelling hierarchies. MCMTs are meant to achieve reusable multilevel model transformations for the specification of behaviour. In this work, we have improved the expressive capabilities of MCMTs by extending them with basic support for attributes, the specification of conditions to block their execution, and the possibility of expressing multiple patterns through the use of nested parametric boxes.

Even though the potential of the MCMTs has been illustrated in several examples, its practical applicability was limited. Indeed, the proposal in [39] was only theoretical and no proper implementation was available. We show here how we have turned the MultEcore editing facilities into a complete development environment in which we can, not only edit our MLM models, but also experiment with them through their simulation and execution, and analyse them by giving access to advanced verification and model checking tools.

We have provided such capabilities for simulation and analysis thanks to a formal specification of MultEcore models in rewriting logic [40,42], and specifically by providing a model-to-model transformation into the rewriting logic language Maude [15,17]. As we will see in the rest of the paper, the syntactical facilities of Maude have allowed us to use a representation of MLM hierarchies and MCMT rules very close to that of MultEcore. Indeed, this minimal representation distance has facilitated the automation of the bidirectional transformation between them. These transformations give MultEcore users access to the Maude execution engine, which is possibly the most efficient engine for rewriting modulo (combinations of) associativity, commutativity and identity [21,18]. In addition, it also gives access to Maude's formal tool environment, which includes, e.g., tools for reachability analysis, model checking, and confluence and termination analysis.

In summary, the contributions of this paper, which extend preliminary work presented in [55], are:

– The MCMTs version described in [39,55] had some practical problems and expressivity limitations. We extend and improve them introducing three main features: (i) attribute definition and manipulation, which brings additional expressivity to the specification of behaviour; (ii) rule conditions that add extra requirements for a rule to be applied; and (iii) nested boxes to handle submodel collections, improving expressiveness and reducing the proliferation of rules. A preliminary and very limited version of these boxes was presented in [55,57]. We present here a fully-operational full-fledged version of them, where boxes may appear in both sides of the rules,

boxes may be nested, and each of them may have an explicit cardinality specified. Basic support for the Object Constraint Language (OCL) [13] has been added for the manipulation of attribute values and for the specification of conditions, which greatly improves the expressiveness of the tool.
- We present in this paper a rewriting logic semantics of MLM hierarchies and MCMT rules through their representation in Maude. This formal representation of MultEcore models allows us to execute and analyse such models using Maude's formal tools.
- A bidirectional transformation between MultEcore MLM models and Maude specifications has been developed. This functional infrastructure connects MultEcore to Maude, allowing us, not only to design our multilevel modelling hierarchy and specify its MCMTs, but also to simulate the specified systems and analyse and verify them using several techniques. Within our infrastructure, we encapsulate Maude as a background process that handles the instructions and return the execution and analysis results, given by the interface that the user uses to interact.
  While there was some basic infrastructure in [55], this is now a mature tool, not only more efficient and configurable, but covering all the features of the language. Although as we discuss in the conclusions section there is much work ahead of us, the MultEcore editor and the transformation between MultEcore and Maude is completely operational.
- The application of the complete infrastructure to a case study for a multilevel DSML for Petri nets, from the design phase to the final execution of the system that we later verify through reachability analysis and model checking techniques. While the case study is described in this paper, we refer the reader to [54] for the complete MultEcore and Maude specifications, including additional details on their analysis.

Outline: We describe in Section 2 the features that characterise our MLM approach using a multilevel DSML for a Petri nets multilevel hierarchy. We level-wise explore each model comprising the hierarchy, from both the structural and behavioural points of views. Section 3 provides an overview of the infrastructure that transforms the multilevel DSML defined in MultEcore into a Maude specification. This section provides details of the generated Maude specification. We demonstrate the use of such an infrastructure with a case study in Section 4, where we perform execution and analysis of a Petri net model of a gas station. In Section 5 we discuss related work. Finally, Section 6 concludes the paper and outlines directions for future work.

2 Multilevel Modelling of Petri nets

The MultEcore tool is designed as a set of Eclipse plugins, giving access to its mature ecosystem (integration with EMF) and incorporating the flexibility of MLM. In the MultEcore approach [39], the abstract syntax is provided by MLM models and the behaviour is provided by Multilevel Coupled Model Transformations (MCMTs) [39,37]. Using the MultEcore tool, modellers can (i) define MLM models using the model graphical editor; (ii) define MCMTs using its rule editor; and (iii) execute and analyse specific models. The execution of MultEcore models rely on a transformation of the models into Maude [15] specifications. When we design a multilevel DSML, we first define its syntax/structure with multilevel modelling hierarchies. Then the behaviour is specified via our multilevel transformation language.

For implementation reasons, MultEcore prescribes the use of Ecore [59] as root graph at level 0 in all example hierarchies. Models are distributed in *multilevel modelling hierarchies*. A multilevel modelling hierarchy in our context is a tree-shaped hierarchy of models with a single root typically depicted at the top of the hierarchy tree. Thus, hierarchies enclose a set of models connected via typing relations. Levels are indexed with increasing natural numbers starting from the uppermost one, having index 0.

To illustrate the different concepts and techniques discussed in this paper, we use as case study a DSML for Petri nets. In the next sections, we describe each of the models that constitute the Petri net multilevel hierarchy. We depict in Appendix A (Figure 19) the complete developed PNs multilevel hierarchy (where we omit Ecore at the top).

2.1 Petri nets metamodel

Petri nets (PNs) is a well-established formalism to model concurrent systems [46,47]. There is a rich body of theoretical results enabling analysis of PNs, and an enormous set of supporting tools.

A PN is a directed bipartite graph, in which the nodes represent transitions (i.e., events that may occur, represented by rectangles/bars) and places (i.e., states, represented by circles). For example, Figure 1 shows a Petri net model using a well-known concrete syntax. In it, we find places `p1`...`p4` and transitions `tr1` and `tr2`, where `p1` and `p2` are connected to `tr1` via input arcs, `p3`

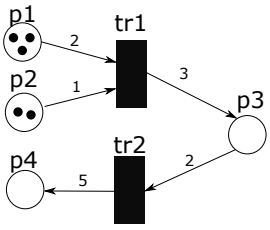

Fig. 1 Simple Petri net model

is connected to tr1 through an output arc and to tr2 via an input arc, and finally p4 is an output place of tr2.

The nodes and arcs constitute the static structure of a PN. The dynamic behaviour of the net is given by the *token game*, representing various states of the system. This token game is based on the firing of transitions that lead to the consumption/production of tokens; each fired transition produces a new model state.

A particular state is a snapshot of the system's behaviour. The state of a place is called its *marking*, represented by the presence or absence of tokens (commonly represented as black dots), in the places. In the example shown in Figure 1 there are three tokens in p1 and two tokens in p2. The current state of the modelled system (marking) is given by the number of tokens in each place.

The increasing complexity of systems has promoted a proliferation of Petri nets variants and extensions during the last decades, as often classical Petri nets are too basic to capture the needs of certain environments. A brief comparison of different kind of Petri nets can be found in [9]. Although our hierarchy could include other types of PNs, here we only include classical or regular PNs and reset/inhibitor nets [65].

We show in Figure 2 a PN metamodel aimed to capture the abstract concepts of Petri nets. This metamodel represents the level 1 of the hierarchy (Figure 19(a)). The purpose of this model is merely structural. In other words, subsequent levels below it should define the concrete semantics of the PN language(s) (as we show in this section). A PN contains nodes, which can be either a Place or a Transition, and Arcs. The tool MultEcore allows us to make use of the *inheritance* relation and to mark Node as an abstract class, which cannot be instantiated (note the italics). As shown in the figure, the type of a node, provided by some element in an upper level metamodel, is indicated in an (light blue) ellipse at its top left side, e.g., EClass is the type of PN, Node, Transition, Place, and Arc. The type of an arrow is written near the arrow in italic font type, e.g., EReference for arcs, nodes, source, target, inArcs and outArcs. We support attribute declarations that can be currently typed by one of the four basic Ecore types, namely Integer, Real, Boolean and String. These attributes can be instantiated in a lower level with a value, as illustrated in Section 2.4. For the manipulation of attribute values, and the specification of rule conditions, a subset of OCL [66,13,12] is currently supported.[1]

The annotations displayed as three numbers in a (red) box at the top right of each node, and concatenated to the name after "@" for every reference, specify their potencies. Potency in attributes is displayed as two numbers as an attribute does not have depth, since first it is declared, and eventually in a level below it is instantiated. The two numbers are specified in front of the attribute name. *Potency* [29] is a well-known concept in MLM and it is used on elements as a way of restricting the levels at which this element may be used to type other elements. By using potencies on elements, we can define the degree of flexibility/restrictiveness we want to allow on the elements of our multilevel hierarchy. The first two values, *start* and *end*, specify the range of levels below, relative to the current level, where the element can be directly instantiated. The third value, *depth*, is used to control the maximum number of times that the element can be transitively instantiated, or re-instantiated, regardless of the levels where this occurs. For instance, the potency specified for Arc, Node, Transition and Place is 1-2-3, which means that an element can be directly instantiated one and two levels below (levels 2 or 3 in the hierarchy), and such instances can be re-instantiated up to 3 additional times. This depth is therefore dependent on the value of the type, and the depth of an element must always be strictly less than the depth of its type.

2.2 Regular Petri nets

The regular Petri nets that we consider in this paper are not restricted to the so-called *Ordinary Petri Nets* where input and output arcs consume or produce, respectively, only a single token [25]. We allow natural numbers on arcs so that more than one token can be added/removed at a time. Following the PNs convention, we denote this number as the *weight* of the arc.

2.2.1 A metamodel for regular Petri nets

Figure 3 displays the *regular-petri-nets* model. It is located at level 2 of the hierarchy (Figure 19) where we instantiate the concepts defined at level 1. Thus, in this

[1] OCL was chosen since it has been part of UML for several years, is one of the most used languages in EMF-based applications, and it is consider a standard in the MDSE community. A full description of the supported subset of OCL, as well as the adaptation of OCL to the multilevel modelling context, will be published elsewhere.

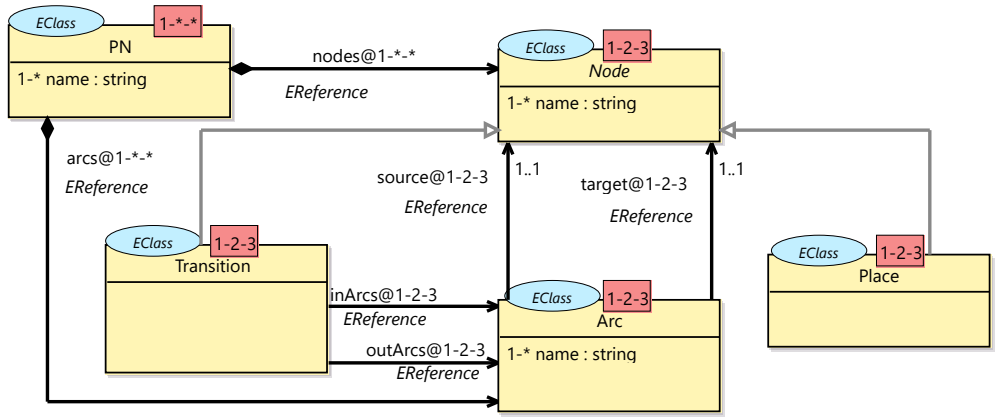

Fig. 2 Conceptual Petri nets metamodel (also shown in Figure 19(a))

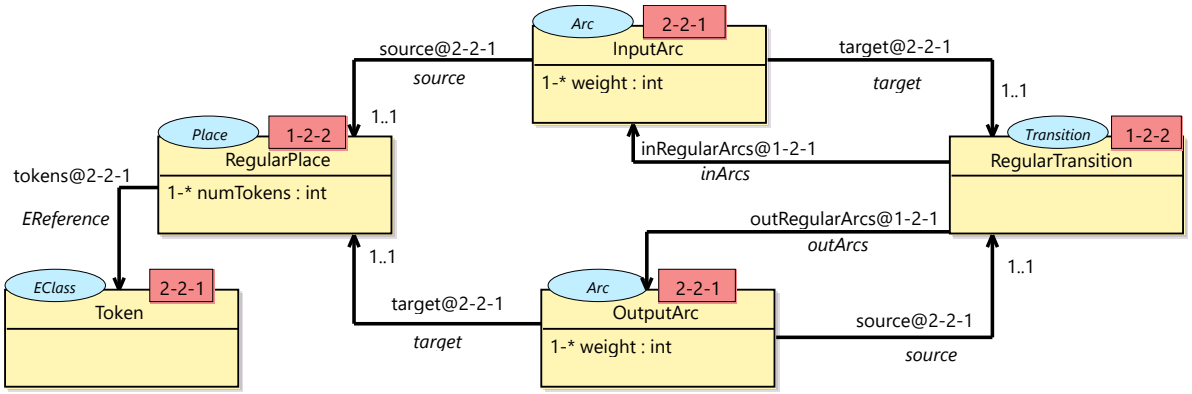

Fig. 3 Regular Petri nets metamodel (also shown in Figure 19(b))

model, we provide the structural basis for the modeller to be able to define further Petri nets instance models. InputArc and OutputArc connect regular places and regular transitions. A RegularPlace controls how many tokens it is holding via the numTokens attribute[2]. The weight of arcs is represented as attributes weight of type int in classes InputArc and OutputArc.

2.2.2 Operational semantics of regular Petri nets

The multilevel transformation language that MCMTs define allows us to exploit multilevel capabilities and is powerful enough to specify behavioural descriptions in an operational way. Transformation rules can be used to represent actions that may happen in the system. A rule has the form of $LHS \Rightarrow RHS$ *if* C, where LHS is a multilevel model pattern (which may contain variables), and RHS is model pattern in which we can use the variables already appearing in LHS.

[2] Please, note that the number of tokens may be calculated with the OCL expression rp.tokens->size(). The attribute is however used to speed up calculations and to illustrate the use of attributes.

C is a boolean condition, in which we can use variables from LHS. Given a model M that represents a state of the system, we say that there is a match of LHS on M if there is a submodel $M|_p$ of M such that for some assignment σ of the variables in LHS, we have $LHS\sigma = M|_p$, where $LHS\sigma$ denotes the application of the assignment σ to LHS. Given a match of LHS on M, for some assignment σ and the submodel $M|_p$, the condition $C\sigma$ is evaluated, where, similarly, $C\sigma$ denotes the application of the assignment σ to the condition C. If it evaluates to *true*, then the application of the rule consists in the replacement of the submodel $M|_p$ of M by $RHS\sigma$, which we denote $M[RHS\sigma]_p$. In other words, if there is a match of the rule on the model, and its condition is satisfied, then the matched submodel is replaced by the model specified in the right-hand side of the rule.

The way to express the behaviour of systems using transformation rules is by specifying rules modelling each of the possible actions that may occur. In PNs, actions occur when transitions are fired. In a regular PN, we only have regular arcs connecting places with transitions. Although transitions can have an arbitrary number of input and output places, such an action can

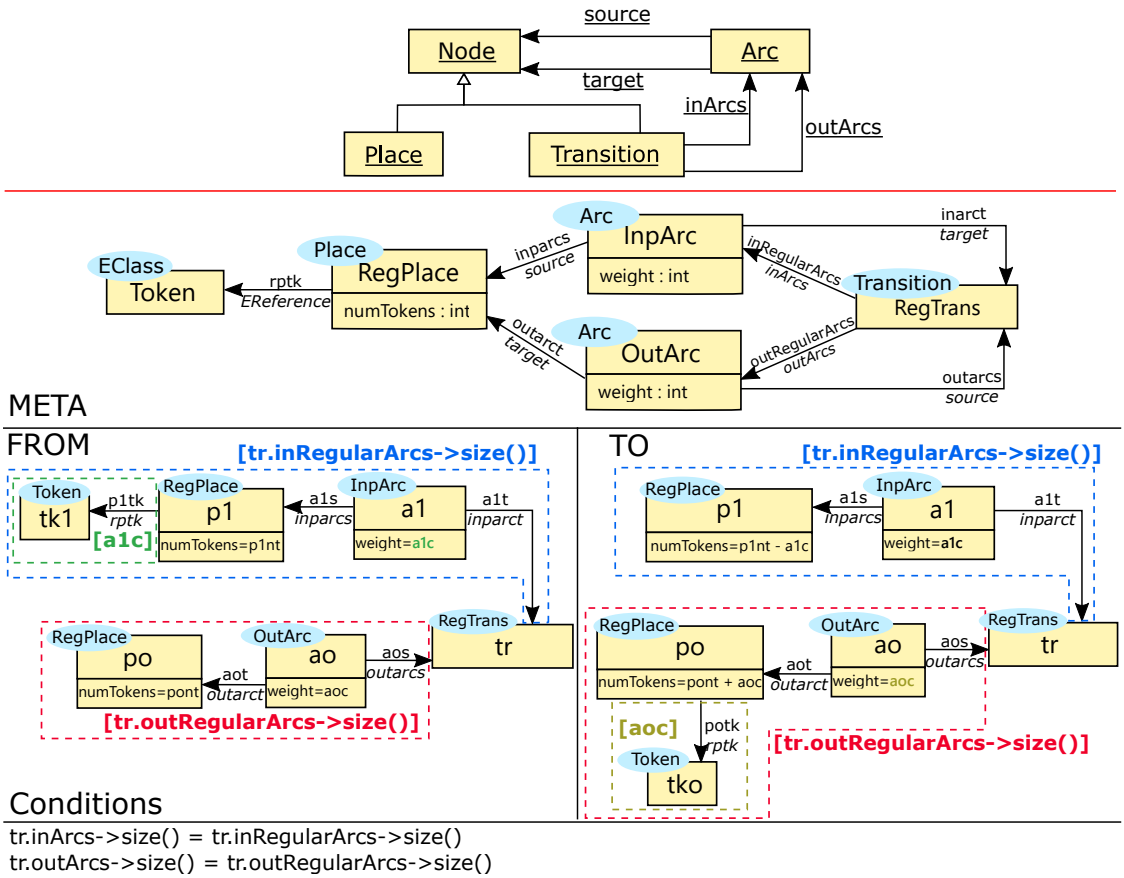

Fig. 4 Rule *Fire regular transition*: It removes tokens in the input places and creates new ones in the output places

be specified with the MCMT rule called *Fire regular transition* depicted in Figure 4. Specifically, this rule models a transition being fired, taking into account the information of the input places (arcs connected to it) and the output places (arcs where new information is to be produced). The FROM and TO blocks describe the left pattern and the right pattern of the rule, respectively. The META block depicts a multilevel pattern allowing us to locate types at any level that can be used as individual types for the items in the FROM and TO blocks, respectively. Notice that the META facilitates the definition of an entire multilevel pattern, therefore, we can specify several META levels within the block.

At the top level of Figure 4, we mirror parts of the *petri-nets-concepts* model (depicted in Figure 2), defining elements like Node, Arc, Transition, Place, source, target, inArcs and outArcs as constants — constant elements have their names underlined and their types are not specified, either via ellipses for nodes or italics text for references. We depict in the META block those elements and their relationships that are useful for the specification of the FROM and TO patterns.

In the second META level (below the red horizontal line), we capture elements to serve as types to scope the execution of regular PNs. In this level we find elements as variables such as Token which type is denoted in the ellipse right above it (EClass), RegPlace of type Place, and the reference rptk of type EReference. Similarly, we express attributes (such as numTokens) that later are going to be used in the levels below.

Please note that the horizontal lines do not enforce consecutiveness between the levels specified in the rule with respect to the hierarchy. This leads to a more natural way of defining that a type is defined at some level above, without explicitly stating at which level. In fact, this also promotes flexibility in case of future modifications of the number of branches (horizontal dimension) and the depth (vertical dimension) of hierarchies. For instance, the three levels depicted in the rule in Figure 4 would match to levels 1, 2 and 4 in the multilevel hierarchy depicted in Figure 19. As the aim of the running PNs multilevel hierarchy is not to highlight the horizontal/vertical flexibility, we refer the interested reader to [55, Section 4.2] for details on this.

We specify in the FROM block what elements must be found in the model in order to be able to fire a transition. As one can observe, dashed boxes are specified around certain parts of the FROM model. A key point

when defining model transformation rules is to make them as reusable as possible. Furthermore, in a PN, there might not only be as many input/output places connected to a transition as one requires, but also an arbitrary number of tokens residing within each of these places. Clearly, it is not practical to define one rule per possible combination of these connections, as the number of rules would rapidly blow up. MCMTs allow the use of nesting boxes to define patterns where its unfolding would result in a collection of elements. As seen in the rule, boxes may appear in both sides, and they can be nested.

The blue dashed box in Figure 4 encapsulates the nodes tk1, p1 and a1, as well as the references p1tk, a1s and a1t, covering all the potential input places connected to the transition (matched to tr) in the model. The number of instances of this pattern submodel is given by the OCL expression tr.inRegularArcs→size(), which represents the number of incoming arcs, i.e., the size of the collection of incoming regular arcs of the transition tr.

In OCL, the size() operator calculates the size of the collection it is applied on. The tr.inRegularArcs expression returns the collection of edges whose source is tr and its type is inRegularArcs. Note, however, that the way in which types are used in MLM is a bit different than for standard OCL. This allows transitive typing, which as we will see below, may be very useful. If instead, as in the condition of the rule, we use tr.inArcs, then we get the collection of edges of type inArcs or any of its instances. Note that the expression tr.inArcs→size() = tr.inRegularArcs→size() checks whether all the incoming arcs of a given transition tr are of type inRegularArcs. This means that the rule is only applicable on transitions whose arcs are all *regular*. The number of total input (resp. output) arcs, inArcs (resp. outArcs), must be equal to the number of input (resp. output) regular arcs, inRegularArcs (resp. outRegularArcs).

Analogously, and using the OCL expression tr.outRegularArcs→size(), a second (red) dashed box allows us to specify a number of output places (and corresponding arcs) connected to the transition.

Note the (green) nested box in the FROM part, inside the (blue) one we were just referring to for the incoming arcs. This inner box allows us to take an arbitrary number of tokens from the input place. For a specific instantiation of the rule, the cardinality of the box is matched to the variable a1c that takes the value of the weight attribute of arc a1. Indeed, given these boxes, a transition may have multiple incoming arcs, and for each incoming place-arc, multiple tokens.

There are also boxes on the TO part. Notice that the input and output arcs are left unmodified, but the appropriate number of tokens are added to the corresponding output places. The number of tokens to put in an output place is provided by the weight attribute of the outcoming arc. The nested (green) box in the TO part, inside the (red) box, indicates that the number of tokens (tko) to be added to each output place po connected to tr via ao, is given by the value aoc of the weight of the arc ao. Finally, note the use of OCL expressions for the manipulation of attributes. In this case, the numTokens attributes of places p1 and po are correspondingly updated: each input place p1 from which some tokens are removed and each output place po that receives tokens, gets its numTokens attribute, respectively, decreased (numTokens = p1nt - a1c) or increased (numTokens = pont + aoc) with the corresponding number of tokens.

In summary, the rule can be executed if the unfolded number of elements is found during the matching process, and all the conditions are satisfied. If this happens, the model in the TO part is produced. In this case, the execution of the rule removes all the tokens present in each of the input places as specified in the boxes, and creates new tokens on the output places.

2.3 Reset/inhibitor Petri nets

A reset/inhibitor PN [65] is a PN that in addition to regular arcs may also have reset and inhibitor arcs. A reset arc is an input arc that connects a place to a transition and that removes all the tokens of the place when the transition is fired. This is useful as a "cleaning mechanism" in models that capture, e.g., certain environments where messages might be retransmitted and buffers could accumulate old messages. An inhibitor arc is an input arc which is used to reverse the logic of an input place. With an inhibitor arc, the absence of a token in the input place is what enables the connected transition (not its presence). For instance, inhibitor arcs can be used to delay certain actions until a system is idle, or to wait until the end of a loop.

Figure 5 shows a very simple example of a reset/inhibitor PN in which we have one arc of each type. In this example, p1 is connected to tr1 via a regular input arc (defined in Figure 3), p2 via a reset arc (denoted with double arrow heads) and p3 via an inhibitor arc (distinguished with a small circle instead of an arrow head). Thus, this transition could be fired according to the semantics of each of the arcs: since (i) p1 has 3 tokens and its regular input arc requires 2; (ii) p2 does not block the firing of the transition, but it will be emp-

tied by its connected reset arc; and (iii) p3 has 0 tokens which fulfils the enabling semantics of the inhibitor arc.

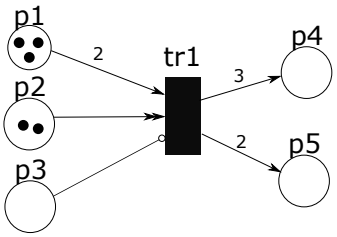

Fig. 5 Concrete syntax of a reset/inhibitor Petri net example

2.3.1 A metamodel for reset/inhibitor Petri nets

Figure 6 shows the model *reset-inhibitor-petri-nets*, placed at level 3 of the hierarchy (see Figure 19(c)). The model captures rules extended with the so-called *reset arcs* and *inhibitor arcs*. As in the model at level 2, the refined Transition in Figure 6 keeps track of the inhibitor and reset arcs connected to it (through references inInhibitorArcs and inResetArcs, respectively). While ExtendedPlace and ExtendedTransition are typed by elements in the level right above, ResetArc and InhibitorArc nodes are typed directly by Arc, which is located two levels above (as denoted by the @2 after the type).

One could argue that these elements that hold specialisation semantics can be realised using inheritance in the metamodel, which indeed is a valid alternative. However, this would lead to a single bigger metamodel where specialisations on elements (e.g., Arc) that belong to different domains are put together and further extensions in each domain would have to be handled in the same metamodel. Although a more detailed discussion may be found, e.g., in [34,6,7], note that, by having this "physical" separation, the modeller has more control on the individual artefacts and therefore the subsequent modifications would be done easier (enhancing reusability). Furthermore, there might be extra horizontal extensions when considering alternative domains, which can be more naturally achieved by promoting this level separation. Note that even we might separate elements within different levels, we do not necessarily make this separation because such elements are related through "type-instance" relationships.

In our approach, we follow the so-called *abstraction semantics* to organise elements within the multilevel hierarchy based on how abstract they are. Thus, for us, organising elements in different models is a feature that primarily enhances modularisation and promotes separation of concerns [6]. In other words, we do not encourage the *level segregation* principle [30], which establishes that level organisational semantics should be unique, i.e., aligned to one particular organisational scheme, such as *classification* or *generalisation*. Nonetheless, we do encourage the *level cohesion* principle [30], that is, we recommend to organise elements that are semantically close (by means of potency and level organisation).

2.3.2 Behaviour for reset/inhibitor Petri nets

Reset/inhibitor Petri nets have additional semantics that have to be properly managed. The MCMT rule *Fire reset/inhibitor transition* is depicted in Figure 7. Please, compare this rule with the *Fire regular transition* rule shown in Figure 4. The rule *Fire reset/inhibitor transition* handles the case in which a transition has any number of arcs of any of the three types (regular, reset or inhibitor), but in particular, if there are only regular arcs, it behaves as the *Fire regular transition* rule. Observe that the rule in Figure 7 includes a third META level, where we capture variable elements such as ExtPlace (of type RegPlace), InhArc (representing inhibitor arcs), ResArc (denoting reset arcs) and ExtTrans. As in the levels above, we determine inResetArcs and inInhibitorArcs references with ExtTrans as source, which can be later used in the OCL expressions for the boxes/conditions.

In the FROM block, we need to specify that we might find any number instances of each of the three kinds of arcs. We do it by encapsulating patterns for each of the arc types into a separate box. Corresponding boxes in the right-hand side specify the corresponding action to take on such an arc and its corresponding place. Notice that boxes for regular and reset arcs have corresponding nested boxes specifying the appropriate number of instances. These boxes are described as follows:

- ***Regular arcs:*** The boxes handling regular arcs are exactly as those depicted in the *Fire regular transition* rule, where the box in the FROM block with cardinality tr.inRegularArcs→size() captures each regular arc a1 connecting a place p1 to the transition tr, and removes the number of tokens of each place as given by the weight a1c on the arc. The corresponding number of tokens is then put in the corresponding output places in the TO block.
- ***Reset arcs:*** The box in the FROM block with cardinality tr.inResetArcs→size() captures the reset arcs a2 that connect input places p2 to the transition tr. To remove all the tokens tk2 present in the connected place p2, the number of tokens in the place is used as cardinality of the inner box. Note that these

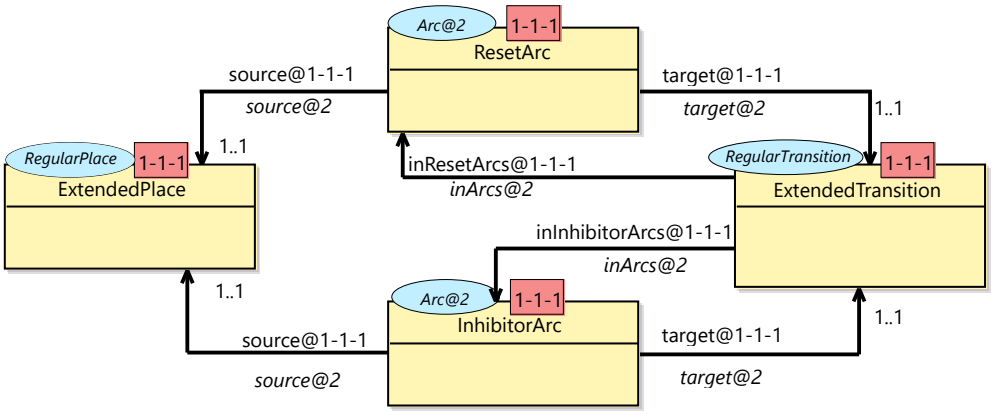

Fig. 6 Reset/inhibitor Petri nets metamodel (also shown in Figure 19(c))

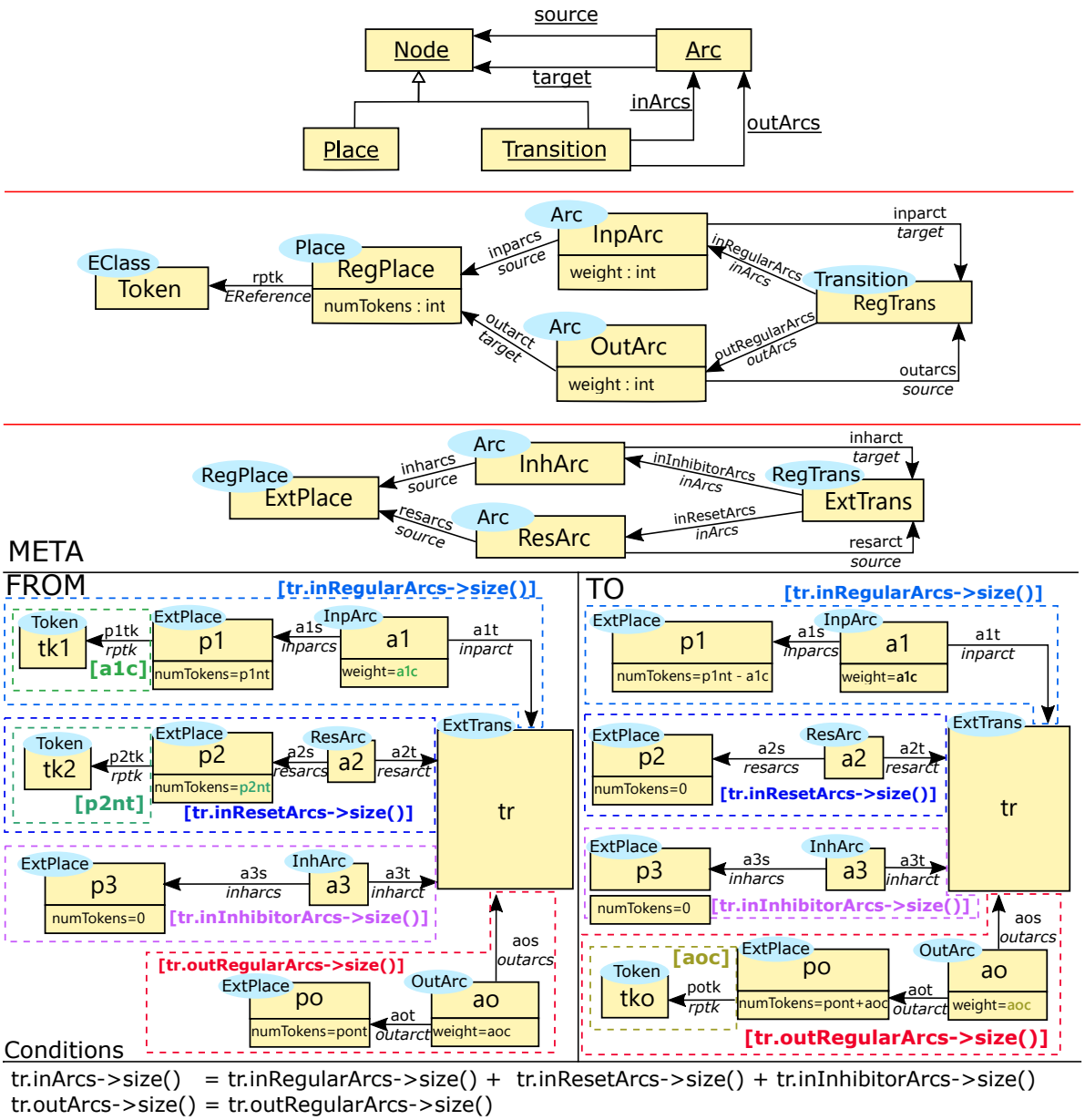

Fig. 7 Rule *Fire reset/inhibitor transition*: modelling the firing of transitions with regular, reset and inhibitor arcs

tokens do not appear in the corresponding box in the TO block. In this way, all of them are removed.

Inhibitor arcs: A third box with cardinality tr.inInhibitorArcs→size() captures inhibitor arcs. Since for the transition to be enabled the number of tokens of each place connected via an inhibitor arc must be 0, we simply specify this directly in p3, where it is stated that the attribute numTokens has value zero.

The rest of the rule looks very similar to what we have already seen. The condition tr.inArcs→size() = tr.inRegularArcs→size() + tr.inResetArcs→size() + tr.inInhibitorArcs→size() checks that the total number of input arcs is the sum of the number of regular input arcs, the reset arcs and the inhibitor arcs. The condition tr.outArcs→size() = tr.outRegularArcs→size() checks that the total number of output arcs is the number of regular output arcs. These conditions would be key for further extensions of the current PN hierarchy.

If the FROM block of the rule matches a submodel of the PN and the conditions are satisfied, the application of the rule results in the removal of the corresponding tokens from the places connected either via regular or reset arcs, and the creation of new tokens in the output places. Notice that the attributes on the places that keep track of the number of tokens get updated.

2.4 Petri nets examples

With the hierarchy described along Sections 2.1–2.3, we can now define models of regular PNs and models of reset/inhibitor PNs. This is possible, as potency specifications allow us to design the hierarchy in a way where *deep instantiation* [5] can be achieved, being able to instantiate elements residing in any level above.

To illustrate how PNs are represented using the given hierarchy, we show a first example using a concrete syntax for Petri nets and then its corresponding one using the MultEcore (abstract) syntax. Figure 8 shows a simple example using regular PNs where four places (two input and two output) and one transition are depicted. To the left we can see that p1 and p2 carry three and two tokens, respectively. Firing tr1 transition would remove 2 tokens from p1 and 1 from p2, and would create 3 and 2 tokens in p3 and p4, respectively, as expressed by the weight in the arcs.

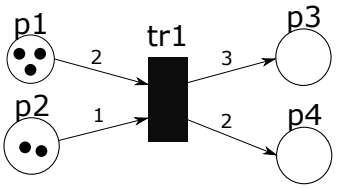

Fig. 8 Concrete syntax of a regular Petri net example

The MultEcore representation of the PN in Figure 8 is shown in Figure 9. Since we consider this model at the instance level, we use potency 0-0-0 in the elements. This is used to enforce that elements at the bottom level (in this case level 4) are used purely as instances, which cannot be refined further at levels below it.

As a second example, the MultEcore representation of the PN depicted in Figure 5 is depicted in Figure 19(d).

3 Execution of Multilevel DSMLs using Maude

Maude [14,15,17] is a specification language based on rewriting logic [41], 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 $(\Sigma;E)$ is an equational theory that specifies the system states as elements of the initial algebra $T_{(\Sigma;E)}$, and R is a set of rewrite rules that describe the one-step possible concurrent transitions in the system. Σ is a signature that specifies the type structure (e.g., sorts and subsorts) and operations, and E is the collection of equations and memberships declared in the functional module. Rewrite specifications thus described are executable, if they satisfy restrictions such as termination and confluence of the equational subspecification, and coherence of equations and rules.

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 grammars. In summary, Maude provides, among others, the following useful features:

Formal specification. The Maude specification of multilevel hierarchies and MCMTs represents a formal semantics of MultEcore models in rewriting logic. Based on such formalisation, the transformation MultEcore $\longleftrightarrow$ Maude has been automated.

Execution of the specification. The Maude specification obtained from MultEcore models using the above transformation is executable, and therefore it can be used to simulate our MultEcore models in Maude. The versatile rewriting engine is not only efficient, but also provides functionalities to customise the way we go through the execution steps. We can simulate our systems by letting Maude choose the path to follow, or we can specify a concrete path specifying it step by step, or by means of execution strategies.

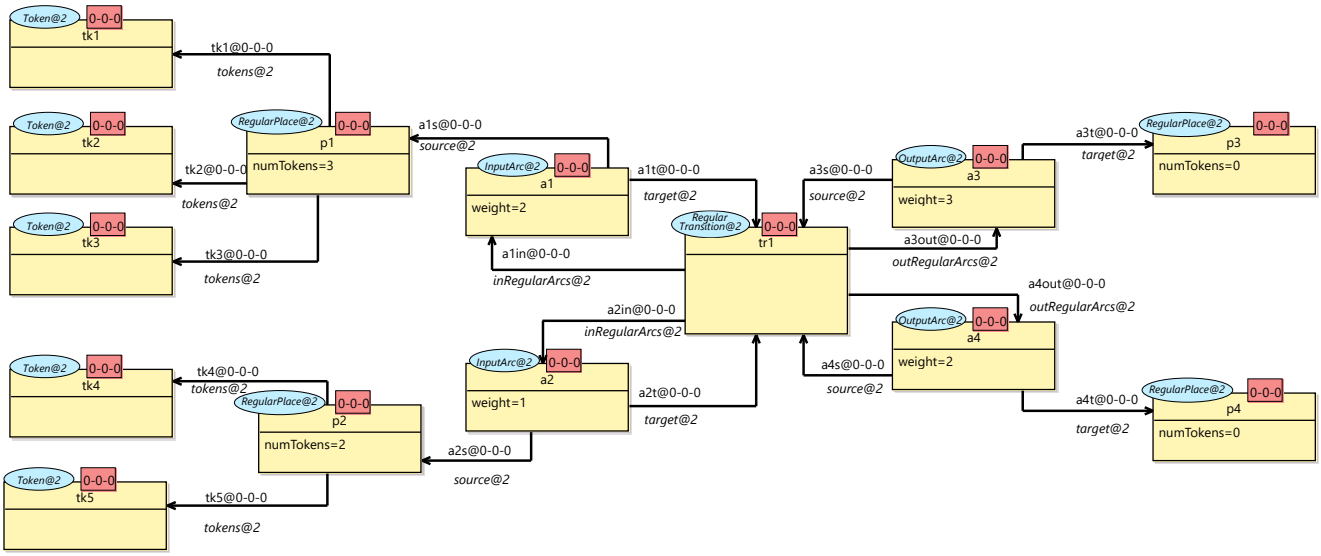

Fig. 9 MultEcore syntax of a regular Petri net example

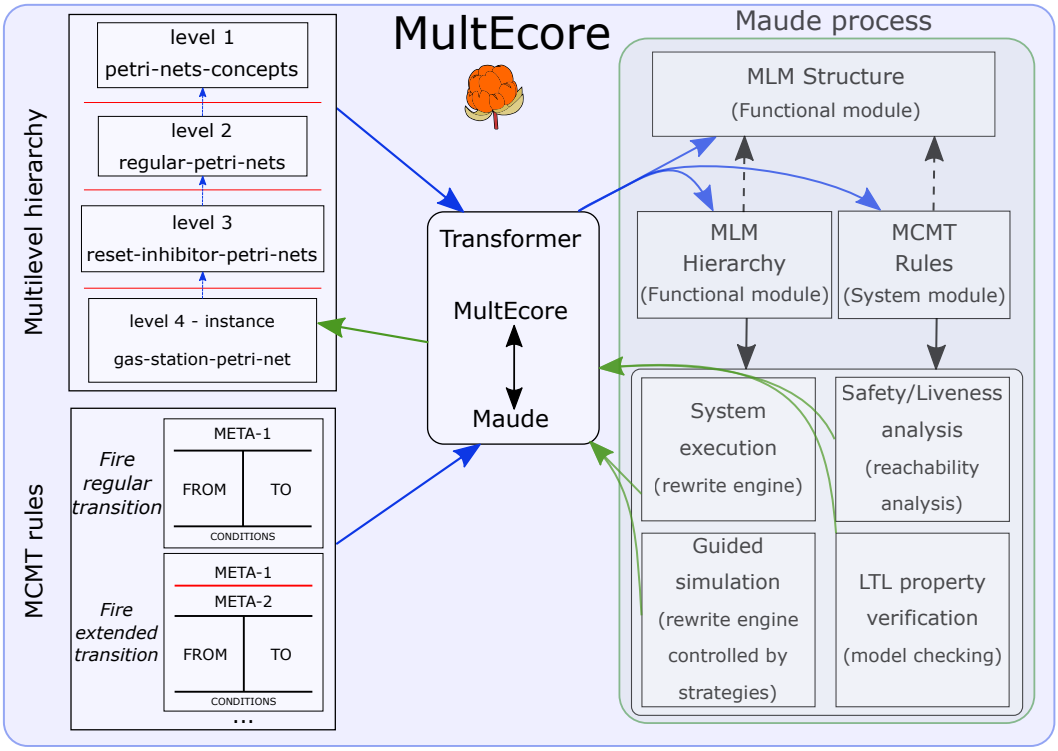

Fig. 10 Infrastructure for the execution and analysis of multilevel modelling hierarchies

Formal environment. Once the rewriting logic specification of a MultEcore model is available, we can use the tools in Maude's formal environment to analyse it. For example, we can check properties such as confluence or termination of our specifications, and can also perform reachability analysis, model checking or theorem proving.

The overall MultEcore-Maude infrastructure is sketched in Figure 10. The left-hand side shows the MultEcore part, where we specify multilevel DSMLs by providing a Multilevel Hierarchy and a set of MCMT rules. The Transformer MultEcore $\longleftrightarrow$ Maude has been developed as a bidirectional transformation that takes MultEcore textual specifications and automatically generates Maude specifications, and then takes the XML output files that Maude produces as result of performing execution and analysis, and automatically

translates them into MultEcore models graphically displayed.

To grasp an intuition of how the transformation works, each MultEcore object (including both a hierarchy and its MCMTs) is mapped into a corresponding Maude object. References and conditions are handled in exactly the same way, by using references as names, and using the same set of expressions (types and operators) for conditions. The rewriting modulo associativity, commutativity and identity available in Maude captures quite naturally the intended operational semantics of MCMTs. The major challenges were the handling of boxing and the performing of the rewriting on multilevel hierarchies. The support for OCL is based on the Maude semantics of OCL proposed in [58].

The right-hand side of Figure 10 shows the Maude process perspective. The transformer produces a functional module with the equational theory used to represent MLM hierarchies, the MLM Hierarchy, and a system module with rewrite theory that represents the MCMT Rules. The representation of MLM hierarchies and MCMTs is presented in Sections 3.1 and 3.2-3.3, respectively. We illustrate in Section 4.1 some of the possibilities for execution and analysis of the models on a case study. As we will see in this section, MultEcore encapsulates the interaction with the Maude tools, which are hidden to the user. The Maude specification is however available to the user, who can interact directly with the Maude environment to get full access to all its features. The complete MultEcore description (both the hierarchy and the MCMTs), the corresponding full Maude specification and the experiments and properties verified can be found in [54].

3.1 Multilevel hierarchies in Maude

In Maude, object-oriented systems can be specified by object-oriented modules in which classes and subclasses are declared, with the usual support for inheritance, and dynamic binding. 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.

To represent multilevel metamodels we have introduced declarations to represent multilevel hierarchies as collections of objects each of which represents one of the level models. Specifically, in our approach, a multilevel hierarchy is represented as a structure of sort System of the form

$$\{\ model_1\ model_2\ \ldots\ model_n\ \}$$

where each $model_i$ is an object of class Model that represents a model in the hierarchy.

Figure 11 shows an excerpt of the Maude specification obtained from the MultEcore Petri net multilevel hierarchy — notice the ellipses added for space reasons. Since levels are numbered starting from 0 (Ecore), the object representing level i's model uses level(i) as identifier. Such an object uses attributes to share the name of the model (name), the name of its immediate metamodel (om), its collection of nodes (elts), and a collection of the relations between these nodes (rels). Elements and relations are themselves represented as objects, of classes Node and Relation, respectively. Each node has attributes to store its name (name), type (type) and its own attributes (attributes). These attributes are again represented as objects with attributes to keep, depending on the level, its name or value (nameOrValue) and its type (type). A relation object has attributes to store its source (source), target (target), and multiplicities, provided by the two usual values (min-mult and max-mult). To avoid name clashes between levels, object identifiers are represented using the operator oid, and nodes and relations using the operator id. Both operators take the level number in which they are defined as first argument, and either a unique number or a string with its actual name.

For instance, the object in lines 1–6 represents level 0, the Ecore model, which has one node with name and type id(0, "EClass") (line 4) and one relation EReference (lines 5–6). Notice how the source and target of this relation refer to the names of the source and target nodes, respectively, which in this case is the same id(0, "EClass"). The *petri-nets-concepts* model in lines 7–14 represents the model in Figure 2 (also Figure 19(a)). Lines 10–12 show the representation of node Node, of type EClass, which has several attribute, among which we can see its attribute with name name of type String. The instance model at level 4 is shown in lines 16–28. Note that among its nodes, there is one with name id(4, "p1") (in lines 20-21), of type id(3, "ExtendedPlace") — a node in its metamodel — which has an attribute of type id(2, "numTokens") with value 0.

3.2 Box-free MCMTs in Maude

In Maude, object-oriented systems are axiomatised by equational theories describing their states as algebraic

```
1   { < level(0) : Model |
2         name : "Ecore",
3         om   : "Ecore",
4         elts : (< oid(0,  1) : Node     | name : id(0, "EClass"),     type : id(0, "EClass"), attributes : none >),
5         rels : (< oid(0,  2) : Relation | name : id(0, "EReference"), type : id(0, "EReference")),
6                                           source : id(0, "EClass"),   target : id(0, "EClass"), ... > >
7     < level(1) : Model |
8         name : "petri-nets-concepts",
9         om   : "Ecore",
10        elts : (< oid(1,  1) : Node     | name : id(1, "Node"), type : id(0, "EClass"),
11                                          attributes : (< oid(1, 2) : Attri | nameOrValue : id(1, "name"),
12                                                                              type : id(1, "String") > )> )>
13                                                       ...),
14        rels : (...) >
15    ...
16    < level(4) : Model |
17        name : "reset-inhibitor-petri-net-example",
18        om   : "reset-inhibitor-petri-nets",
19        elts : (...
20                < oid(4, 12) : Node     | name : id(4, "p1"),  type : id(3, "ExtendedPlace"),
21                                          attributes : (< oid(4, 13) : Attri | nameOrValue : 0, type : id(2, "numTokens") >) >
22                ...
23                < oid(4, 28) : Node     | name : id(4, "tk1"), type : id(2, "Token"), attributes : none >
24                ...),
25        rels : (...
26                < oid(4, 36) : Relation | name : id(4, "tk1"),  type : id(2, "tokens"),
27                                          source : id(4, "p1"), target : id(4, "tk1"), ...>
28                ...) > }
```

Fig. 11 Excerpt of the Petri net multilevel hierarchy in Maude representation

data types and collections of conditional rewrite rules specifying their *behaviour*. Rewrite rules are written as

$$\mathsf{crl}\ [l] : T \Rightarrow T'\ \mathsf{if}\ C$$

where l is the rule's label, T and T' are terms, and C is its guard or condition. As MultEcore's MCMTs, Maude 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'. Also as for MCMTs, 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.

Given the representation of multilevel hierarchies presented in the previous section, the transformation of MCMT rules without boxes is straightforward. Basically, since the META section does not change, its corresponding representation appears in both left- and right-hand sides. The left-hand side of the rule is completed with the representation of the FROM block, and the representation of the TO block is added to its right-hand side. Variables in MCMTs are represented as Maude variables, and conditions are placed in the conditions as such. Since we restrict conditions to basic types, basic operators and certain selected OCL operations, the expressions can be handled directly by Maude. The last issue to consider is new names and identifiers in right-hand sides. As we explained above, identifiers are represented using the id and oid operators, which take the level in which the object is created and a unique number as arguments. Such numbers are generated using a Counter object, whose value attribute gets increased every time a new identifier is created.

Let us illustrate this general procedure on a specific example. Consider the rule *Fire regular transition* depicted in Figure 4, but let us assume first that it has no boxes in it, that is, let us assume that it takes one single token from the unique input place of a transition and moves it to its unique output place. If this were the case, the corresponding generated Maude rule would be the one shown in Figure 12. Again, notice the ellipses.

The left- and right-hand sides of the rule are given in lines 16–34 and 39–58, respectively. The corresponding condition is in lines 61–62. The META section is represented in lines 2–15 and 37–38. The three objects representing the META section are replicated in both sides. They provide the appropriate context for the rule, but it is in the FROM and TO sections where the actual change is modelled. Notice the use of variables to identify the levels (L1, L2 and L3). These are used to match any specific levels in the hierarchy on which the rule is applied. They do not need to be consecutive levels, the only restriction is given in the condition, where it is checked that $\mathsf{L1} < \mathsf{L2} < \mathsf{L3}$ (line 61). Notice that the Maude rule represents quite closely the corresponding MultEcore MCMT. For instance, constants in the MCMT rules are mapped into Maude constants and ground terms. Variables are used both to represent free elements in the rules and also any other elements not explicitly specified. The condition of the MCMT rule is written as such in line 62. Finally, notice the use of the Counter object, which in the left-hand side has some value N (line 35) and in the right-hand side has

```
crl [Fire-1-to-1-Regular-Arcs] :
   { < level(L1) : Model | name : M,
        elts : (< O01 : Node | name : id(L1, "Arc"), type : id(0, "EClass"), Atts01 >
                < O02 : Node | name : id(L1, "Node"), type : id(0, "EClass"), Atts02 >
                < O03 : Node | name : id(L1, "Transition"), type : id(0, "EClass"), Atts03 >
                < O04 : Node | name : id(L1, "Place"), type : id(0, "EClass"), Atts04 >
                Elts),
        rels : (
                < O05 : Relation | name : id(L1, "inArcs"), type : id(0, "EReference"), source : id(L1, "Transition"), Atts05 >
                < O06 : Relation | name : id(L1, "outArcs"), type : id(0, "EReference"), source : id(L1, "Transition"), Atts06 >
                < O07 : Relation | name : id(L1, "source"), type : id(0, "EReference"), source : id(L1, "Arc"), Atts07 >
                < O08 : Relation | name : id(L1, "target"), type : id(0, "EReference"), source : id(L1, "Arc"), Atts08 >
                Rels),
        Atts >
     < level(L2) : Model | ... >
     < level(L3) : Model | name : M''',
        elts : (< O26 : Node | name : tr_1, type : id(L2, "Transition"), Atts26 >
                < O27 : Node | name : ao_1, type : id(L2, "OutputArc"),
                  attributes : (< O28 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts28 > Attri27), Atts27 >
                < O29 : Node | name : po_1, type : id(L2, "Place"),
                  attributes : (< O30 : Attri | nameOrValue : pont, type : id(L2, "numTokens"), Atts30 > Attri29), Atts29 >
               < O31 : Node | name : tk1_1, type : id(L2, "Token"), Atts31 >
                < O32 : Node | name : a1_1, type : id(L2, "InputArc"),
                  attributes : (< O33 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts33 > Attri32), Atts32 >
                < O34 : Node | name : p1_1, type : id(L2, "Place"),
                  attributes : (< O35 : Attri | nameOrValue : p1nt, type : id(L2, "numTokens"), Atts35 > Attri34), Atts34 >
                Elts'''),
        rels : (< O36 : Relation | name : a1t_1, type : id(L2, "target"), source : a1_1, target : tr_1, Atts36 >
               < O37 : Relation | name : p1tk_1, type : id(L2, "tokens"), source : p1_1, target : tk1_1, Atts37 >
                < O38 : Relation | name : a1s_1, type : id(L2, "source"), source : a1_1, target : p1_1, Atts38 >
                < O39 : Relation | name : aos_1, type : id(L2, "source"), source : ao_1, target : tr_1, Atts39 >
                < O40 : Relation | name : aot_1, type : id(L2, "target"), source : ao_1, target : po_1, Atts40 >
                Rels'''),
        Atts''' >
     < counter : Counter | value : N >
     Conf }
=> { < level(L1) : Model | ... > ---- as in the left-hand side
     < level(L2) : Model | ... > ---- as in the left-hand side
     < level(L3) : Model | name : M''',
        elts : (< O26 : Node | name : tr_1, type : id(L2, "Transition"), Atts26 >
                < O27 : Node | name : ao_1, type : id(L2, "OutputArc"),
                  attributes : (< O28 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts28 > Attri27), Atts27 >
                < O29 : Node | name : po_1, type : id(L2, "Place"),
                  attributes : (< O30 : Attri | nameOrValue : pont + 1, type : id(L2, "numTokens"), Atts30 > Attri29), Atts29 >
                < O31 : Node | name : a1_1, type : IA_1,
                  attributes : (< O32 : Attri | nameOrValue : 1, type : id(L2, "weight"), Atts32 > Attri31), Atts31 >
                < O33 : Node | name : p1_1, type : id(L2, "Place"),
                  attributes : (< O34 : Attri | nameOrValue : p1nt - 1, type : id(L2, "numTokens"), Atts34 > Attri33), Atts33 >
               < oid(L3, N) : Node | name : id(L3, N + 1), type : id(L2, "Token"), attributes : none >
                Elts'''),
        rels : (< O36 : Relation | name : a1t_1, type : id(L2, "target"), source : a1_1, target : tr_1, Atts36 >
                < O38 : Relation | name : a1s_1, type : id(L2, "source"), source : a1_1, target : p1_1, Atts48 >
                < O39 : Relation | name : aos_1, type : id(L2, "source"), source : ao_1, target : tr_1, Atts39 >
                < O40 : Relation | name : aot_1, type : id(L2, "target"), source : ao_1, target : po_1, Atts40 >
               < oid(L3, N + 2) : Relation | name : id(L3, N + 3), type : id(L2, "tokens"),
                                             source : po_1, target : id(L3, N + 1), min-mult : 1, max-mult : 1 >
                Rels'''),
        Atts''' >
     < counter : Counter | value : N + 4 >
     Conf }
if L1 < L2 /\ L2 < L3
/\ tr . inArcs -> size() = tr . inRegularArcs -> size() /\ tr . outArcs -> size() = tr . outRegularArcs -> size() .
```

Fig. 12 Excerpt of the Maude rewrite rule corresponding to the box-free version of the *Fire regular transition* MCMT rule

value N + 4 (line 59) since four new identifiers are introduced.

The model changes applied by the rule have been framed to ease its comprehension. Specifically, given a transition tr_1 (line 17) with only one place (notice the weight 1 of the input arc in lines 23–24), and given one of the tokens in it (the token is specified in line 22 and the relation associating it to the place in line 29), the rule removes such a token and creates a new one (lines 49 and 55–56). The number of tokens in the input place is decremented (lines 26 and 48) and the number of tokens in the output place is incremented (lines 21 and 44). Finally, the created token and relation objects (lines 49 and 55–56) have identifiers oid(L3,N), oid(L3,N+1), oid(L3,N+2), and oid(L3,N+3).

3.3 MCMTs in Maude

As illustrated with the rules depicted in Figures 4 and 7, boxes allow us to express very general situations in a quite intuitive way. However, Maude does not provide any mechanism similar to that of MCMT boxes, and therefore the transformation is not as simple. To handle boxes in a generic and efficient way, we use Maude's meta-programming capabilities to unfold boxes at runtime as needed.

If we look at the *Fire regular transition* rule in Figure 4, this time considering its boxes, we know that each time the rule is applied, depending on the specific situation, there will be a number of replicas of each of the boxes. Actually, notice that we may have multiple boxes in both sides, with different cardinalities, and we can have nested boxes, as many times as needed. Note also that these cardinalities are explicitly specified, otherwise, for example, a given transition could be applied taking an arbitrary number of tokens from several of the available input places. These cardinalities could be provided as OCL expressions, which need to be evaluated to get the corresponding value at the time it is required. In this particular case, the transition has tr.inRegularArcs->size() input regular arcs, each of which has a weight a1c which specifies the number of tokens to be removed from it when the transition is fired. Similarly, the transition has tr.outRegularArcs->size() out regular arcs, which tell us the number of output places, each of which has a weight that indicates the number of tokens to be created on that place.

The only assumption that we make to handle boxes is that their cardinality must be greater than zero. In case we want to consider the possibility of zero replications of a box, we need to provide the corresponding rule without such a box. This is the case for the rule in Figure 7. The cases in which we have transitions with no reset, inhibitor or regular arcs must be handled in different Maude rules. Although these cases can be handled by automatically creating the zero-case corresponding rule, we focus here on the general case.

An MCMT rule with boxes produces two Maude rules (plus the corresponding ones for the zero-cardinality cases). Excerpts of the two rules for the MCMT rule *Fire regular transition* in Figure 4 are shown in Figures 13 and 14. The first one of the rules has a left-hand side as if there were no boxes in it. That is, the left-hand side of the rule in Figure 13 is exactly as the left-hand side of the rule in Figure 12. In its condition, the boxes are expanded in a copy of the rule in accordance with the actual match and its application is attempted (lines 15-24 in Figure 13). If such an application succeeds, the result is given as result of the application of the rule, that is, it is used to replace the current system (line 7). If the application in the condition fails, the rule fails. To understand this rule, we need to introduce some additional Maude machinery and some auxiliary functions.

First, in addition to equality checks, Maude rule conditions may include so-called matching equations using the operator :=. Given a pattern term P (a canonical term possibly with free variables) and a term T which may use variables in the left-hand side of the rule and also variables introduced in previous matching conditions, the condition expression P := T evaluates the term T and tries to match its result to the pattern P. If P is a variable, it works like a *let* or *where* clause to assign that value to the variable so that it can later be used. If a more general pattern is used, the match may result in the simultaneous assignment of values to multiple variables. In the rule in Figure 13, we can see how matching conditions are used several times. First, it is used to refer to the left-hand side of the rules as { Conf' } (in lines 9-13), then to refer to the match of the rule as Subst, and finally to get the result of the application of the unfolded rule (line 15).

In Maude, terms and modules have a metarepresentation that we can manipulate as regular terms. Up and down functions allow us to move terms and modules between levels. For instance, given the module GENERIC-PN in which these rules are defined, the expression upModule('GENERIC-PN, false) gives us its metarepresentation. Similarly, upTerm({ Conf' }) gives us the metarepresentation of the term { Conf' }, and downTerm(T, { none }) moves down the result of the application of the unfolded rule T. The built-in function metaApply(M,T,L,S,N) returns the Nth solution of applying rule L in module M on term T using the substitution S to constraint the application of the rule. Then, assuming that the makeModule function takes

```
 1 crl [FireRegularArcs] :
 2   { < level(L1) : Model | ... >              ---- left-hand side as for the rule without boxes
 3     < level(L2) : Model | ... >
 4     < level(L3) : Model | ... >
 5     < counter : Counter | ... >
 6     Conf }
 7  => downTerm(T, { none })
 8  if (tr . outArcs -> size() = tr . outRegularArcs -> size()) /\ (tr . inArcs -> size() = tr . inRegularArcs -> size())
 9  /\ Conf' := < level(L1) : Model | ... >
10              < level(L2) : Model | ... >
11              < level(L3) : Model | ... >
12              < counter : Counter | ... >
13              Conf
14  /\ Subst := ( ... )                         ---- match of the rule
15  /\ { T, Ty, Subst' } := metaApply(
16                            makeModule(
17                              upModule('GENERIC-PN, false),
18                              upTerm({ Conf' }),
19                              'FireRegularArcsBoxes,
20                              Subst),
21                            upTerm({ Conf' }),
22                            'FireRegularArcsBoxes,
23                            Subst,
24                            0) .
```

Fig. 13 Excerpt of the first of the Maude rewrite rule corresponding to the *Fire regular transition* MCMT rule

```
 1 rl [FireRegularArcsBoxes] :
 2    { < level(L1) : Model | ... >     ---- as the left-hand side of the rule without boxes
 3      < level(L2) : Model | ... >
 4      < level(L3) : Model | ... >
 5      boxes((tr . outRegularArcs -> size()){ 027, 029, 039, 040 },                    ---- box information
 6              box[tr . inRegularArcs -> size()]{ 032, 034, 036, 038, box[a1c]{ 022, 029 } } ))
 7      < counter : Counter | value : N >
 8      Conf }
 9 =>
10    { < level(L1) : Model | ... >    ---- as the right-hand side of the rule without boxes
11      < level(L2) : Model | ... >
12      < level(L3) : Model | ... >
13      boxes(( box[tr . outRegularArcs -> size()]{ 027, 029, 039, 040, box[aoc]{ oid(L3, N), oid(L3, N + 2)} },
14              box[tr . inRegularArcs -> size()]{ 032, 034, 036, 038 } ))
15      < counter : Counter | value : N + 13 > Conf } .
```

Fig. 14 Excerpt of the second of the Maude rewrite rule corresponding to the *Fire regular transition* MCMT rule

the module with the rules and expands the boxes of the indicated rule as required, the call to metaApply in the last matching condition in the rule in Figure 13 will apply the expanded rule on the current state of the system using the original substitution, that is, forcing the same match. The metaApply functions gives a triple {T, Ty, Subst} as result, where T is the term resulting from the application of the rule, Ty its type, and Subst is the complete substitution used in the application.

The FireRegularArcsBoxes rule is shown in Figure 14. It is similar to the rule without boxes explained in Section 3.2, but notice that it also includes information on the boxes and the level 3 object in the RHS which now represents the TO part (line 12).

Boxes are specified as a collection of terms of the form box[C]{OS}, with C being the cardinality of the box and OS the set of identifiers of the objects (nodes and relations) and nested boxes within the box. The makeModule function is a metalevel function that operates on the metarepresented module, expanding the indicated rule by unfolding the boxes in it. It proceeds recursively, removing one box level at a time. As we have seen above, the cardinality of a box specifies the number of replicas of that box that we need to generate. After a box is expanded, the cardinalities of the next-level boxes can be evaluated. The operation is repeated until no further boxes are left. Once all boxes in a rule are completely expanded, the application of the rule is attempted in one single step.

Notice that boxes in right-hand sides may, and in fact do in the FireRegularArcsBoxes rule in Figure 14, contain identifiers of new objects. Notice also that the counter object is updated according to the number of objects being created, either inside or outside boxes.

4 Execution and Analysis of Models

The capability to execute MultEcore systems and use the powerful tools Maude implements for reachability and model checking of the multilevel models takes our infrastructure to a next step. Furthermore, modelling behavioural languages, such as Petri nets, gets interesting if one can transfer simulation and analy-

sis onto the concrete system models. We use, as case study to demonstrate these capabilities, a gas station model adapted from [26]. We consider the whole modelling cycle, where the modeller sketches and designs the multilevel hierarchy that represents the system, then specify the behaviour by means of transformation rules, and then automatically transforms its setting to Maude where simulation and execution can be done to later verify and analyse the obtained system.

The concrete syntax of the model is depicted in Figure 15. This model would be located at level 4 of the PN hierarchy, as an instance of the *reset-inhibitor-petri-nets* model (Figures 6 and 19(c)). The PN model represents a system in which car tanks get filled up at a gas station. The station has a tank with a maximum capacity, which can be evacuated for cleaning reasons and then replenish. If there is no car in the station, a new car can arrive and set its indicator on. Once the car's tank is filled, it leaves the station.

The initial marking of the model, depicted in Figure 15, has 4 tokens in the place Station Tank, which is its full capacity. For a car tank to be refuelled, there must be a token in the Fuel Indicator On place. This can only happen if the transition Turn Fuel Indicator On has been fired, and for this to happen, there cannot be any token neither in the Car Tank place (the tank is empty) nor in the Fuel Indicator On place (the indicator is off). This is modelled by the two inhibitor arcs connected to the Turn Fuel Indicator On transition. Then, once the indicator has been turned on, we can only progress by the firing Fuel Car transition, which makes the Car Tank to be filled (i.e., gets one token). Ultimately, once the car tank is full, the Leave Station transition can be fired, leading the car to exit the station by putting a token into the Outside Station place.

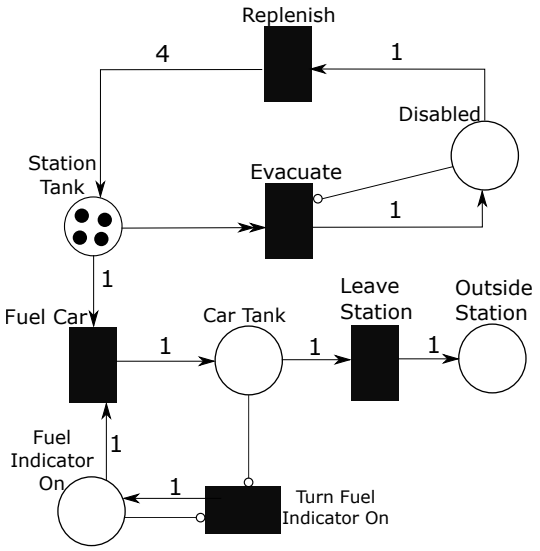

Fig. 15 Gas station model with initial marking

4.1 Execution and analysis using Maude

Given a model with an initial marking, we can simulate it. In fact, we have two ways of doing so. We can let the default strategy choose the rules to apply, and the way in which to apply them, or we can force a specific sequence of rule applications.

Rule rewriting is a highly non-deterministic process, and in general, at every step many rules could be applied. Moreover, since a rewrite system may be non-terminating, as is the case of the gas station example, a maximum number of rewriting steps to be taken may be specified.

A finer control on rule application may sometimes be desirable. In MultEcore, we may specify the sequence of rules to be applied, which can be selected from a list of possible ones. Let us show an example. But first, as we said in Section 3.3, for any rule with boxes, several Maude rules are generated, corresponding to the different combinations of boxes with cardinality zero. Thus, for the *Fire reset/inhibitor transition* rule in Figure 7, seven rules are generated, for transitions with no regular arcs (FireResetInhibitorArcs), for transitions with no inhibitor and no reset arcs (FireRegularArcs), for transitions with no reset and no regular arcs (FireInhibitorArcs), etc. Then, we can specify the strategy with which we desire to rewrite our initial marking model by indicating the corresponding sequence of rule labels. For example, we can guide the execution from an initial marking given the following sequence:

```
FireInhibitorArcs   --------- Fuel Indicator On
FireRegularArcs     ---------- 1st car fuelled (Car Tank)
FireRegularArcs     ---------- 1st car leaves (Outside Station)
FireInhibitorArcs   --------- Fuel Indicator On
FireRegularArcs     ---------- 2nd car fuelled (Car Tank)
FireRegularArcs     ---------- 2nd car leaves (Outside Station)
FireResetInhibitorArcs ---- Disabled
FireRegularArcs     ---------- Station tank filled
```

The comments on the right indicate the corresponding effect on the model. The MultEcore integration with Maude allows us to specify sequence of rules to automatically generate the desired model state and get its graphical representation right away. A screenshot of the MultEcore tool is depicted in Figure 17, in which we can see how rule labels are selected from a list corresponding to, the above sequence of rules. Note that the MultEcore syntax of the initial state (depicted in Petri nets syntax in Figure 15) is in the background

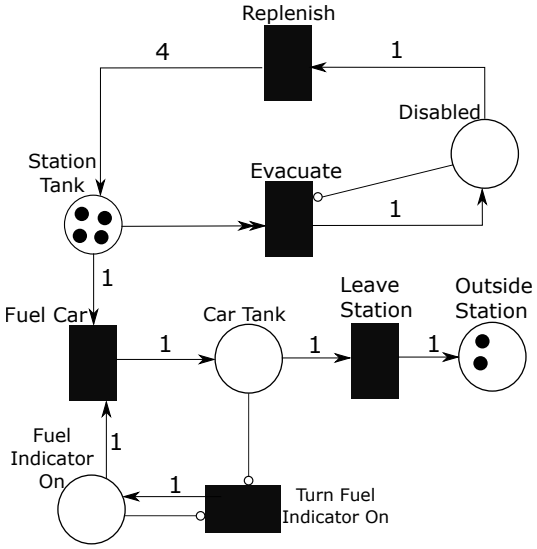

Fig. 16 Resulting Gas station state

of Figure 17. Pressing Finish on such a wizard automatically provides the model state in the MultEcore syntax, shown in Figure 18, where we have four tokens (token19211...token19214) in the Station Tank place, representing that the station tank has been replenished (highlighted at the top-left of Figure 18), and four tokens in the Outside Station place, representing that 4 cars have left the station (highlighted at the bottom-right of Figure 18).

4.2 Reachability analysis

To perform a more exhaustive verification of the model, we can perform reachability analysis and bounded model-checking of invariants, with which we can check safety properties. Specifically, we can study the reachability of given states using the *search* command, where the states to check can be specified both using patterns or conditions on the states. The search command explores the reachable state space following a breadth-first strategy.

To carry on a search, we need to provide: (i) the model from which to initiate the search, (ii) the maximum depth of the search (even for terminating systems, a search may take a long time), (iii) the pattern model to be reached (a model with variables), and (iv) an optional property that has to be satisfied by the reached state. Since the structure of the PN does not change along the execution, we can specify our pattern model leaving as variables the tokens in each place. Then, the condition to satisfy at the target state may be specified as an OCL expression. As result, MultEcore will determine whether such a state is reachable, in the specified number of steps, and if so, it may provide the specific path leading to the state found.

Let us see how we can use the search command to verify properties on the gas station example. For example, we can verify that, starting from the marking depicted in Figure 15, that the system can reach a marking where four cars have left the station and the station tank is again full to continue fuelling further cars. To do that, we just need to select the initial model, the target pattern model, and write, for example, the following OCL boolean expression:

```
id(4, "Station Tank").tokens->size() = 4
and id(4, "Outside Station").tokens->size() = 4
```

MultEcore responds positively, and provides the sequence of rule names leading to such a solution:

```
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireInhibitorArcs
FireRegularArcs
FireRegularArcs
FireResetInhibitorArcs
FireRegularArcs
```

Notice that this is the path to one of the possible states satisfying this condition, which, like in this case, may be not unique.

We can also check whether certain miss-behaviours may occur. Below we list properties we have verified in order to assure the correctness of the model:

- **Property 1.** It is not possible to find a state where, simultaneously, the places Station Tank and Disabled contain the tokens. This would imply that either Evacuate or Replenish transition could be fired more than once in a row, which is not the behaviour we expect for this specific system. We can check this property using the same initial and pattern markings as before together with the following property:

  ```
  id(4, "Station Tank").tokens->size() > 0
  and id(4, "Disabled").tokens->size() > 0
  ```

  Since the system is not terminating, and we expect not to find it, we specify a bound, for example, of 20. The answer from MultEcore is 'false', indicating that such a state cannot be reached within the given depth.
- **Property 2.** It is not possible to find a state where either the Fuel Indicator ON, Car Tank, and Disabled places have more than one token. In other words,

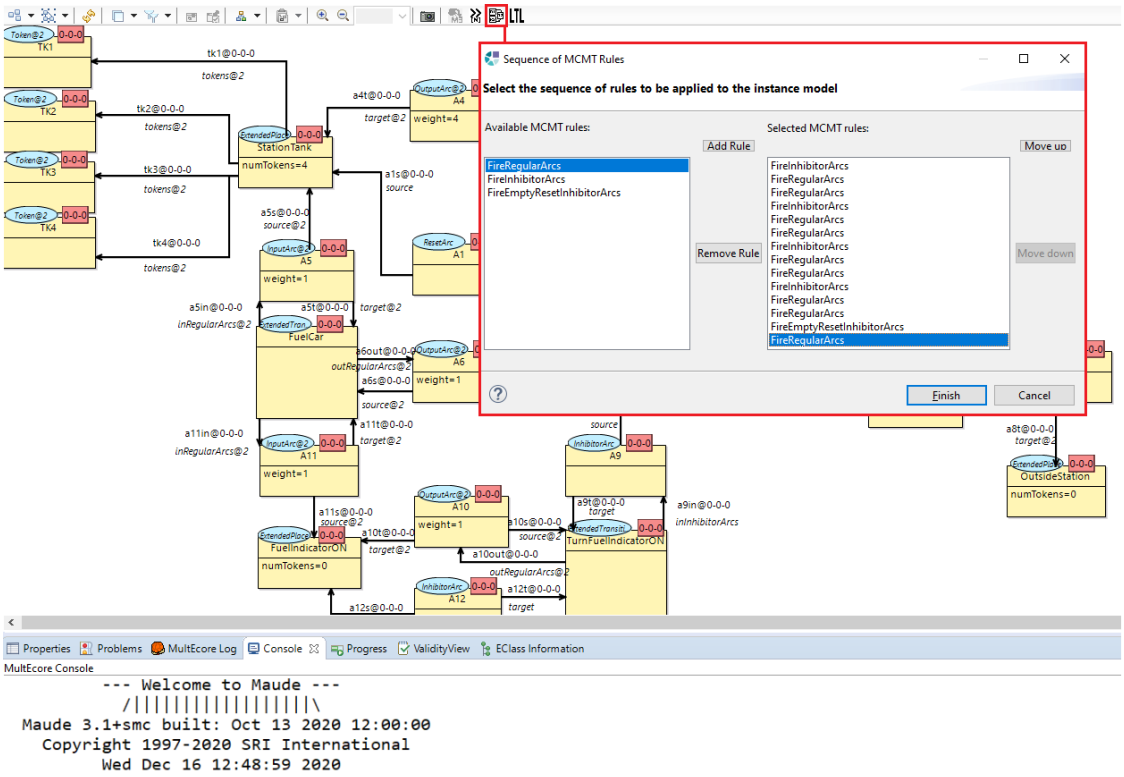

Fig. 17 MultEcore screenshot for sequence of rules-based execution

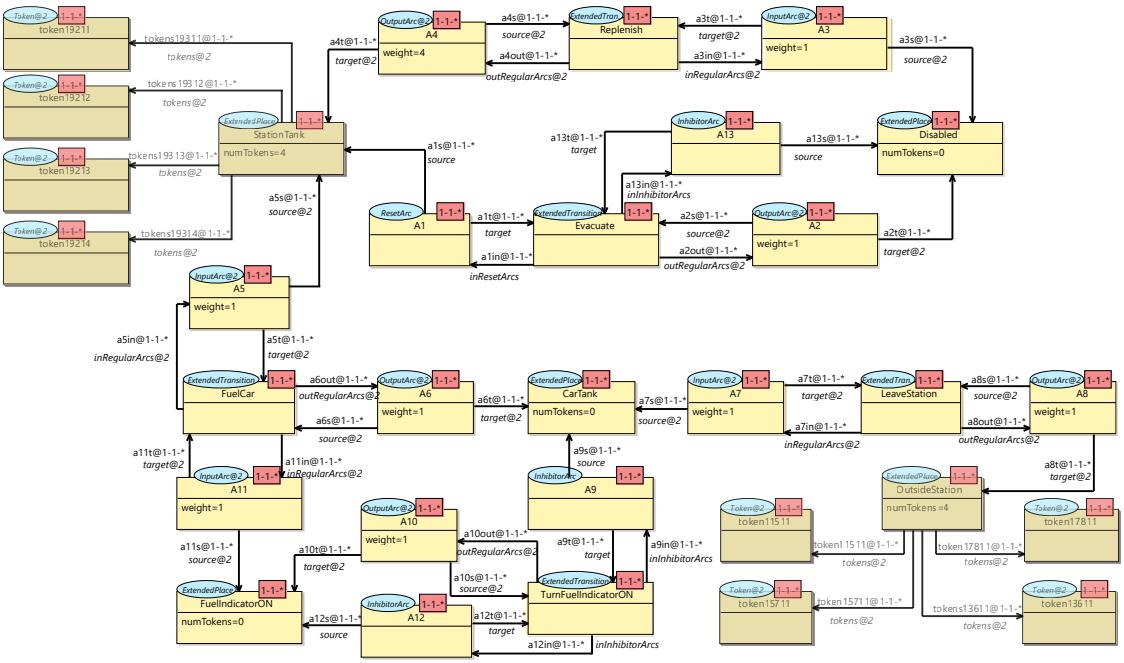

Fig. 18 MultEcore syntax of the Gas station model state after execution

the indicator is either ON or OFF, the car tank is either full or not, and we cannot disable the station consecutively two times. This property can be verified as the previous one using the following OCL expression:

```
id(4, "Fuel Indicator ON").tokens->size() > 1
or id(4, "Car Tank").tokens->size() > 1
or id(4, "Disabled").tokens->size() > 1
```

Again, for a bound of 20, the answer from MultEcore is 'false'.

– **Property 3.** We can find a state where 5 cars have successfully exited the station. If 5 cars can exit the station, then any number of cars can leave as well.

```
id(4, "Outside Station").tokens->size() > 5
```

The answer obtained is positive, and the path to the first found solution is provided.

All the properties listed above have provided the expected results, which further validates our model.

4.3 System abstraction for unbounded analysis

In the previous section we have carried out bounded model checking of several behavioural properties. Although limited to a maximum depth, the verification of the properties checked using the search command greatly increase the confidence in the correctness of the system. However, bounded model checking is an incomplete procedure, since a counterexample could exist at greater depth. The problem in this case is that the state space is infinite, and therefore, we cannot complete the analysis in this way. One way to fully verify the system is using a finite-state abstraction of it, that is, on an appropriate quotient of the original system whose set of reachable states is finite. The method proposed in [43] creates an abstraction of the original system by adding a set of equations to collapse the infinite set of reachable states into a finite set. The specification, extended with these equations, need to still satisfy the usual executability conditions — the equations must be ground Church-Rosser and terminating, and the rules should be ground coherent with them — but the procedure is quite simple, and the abstraction is then correct by construction. The method is valid both for the verification of invariants and LTL formulas with an additional deadlock-freedom requirement. Indeed, an automatic procedure to complete specifications so that the requirement is satisfied has been given in [15].

The key idea about abstraction for invariant verification is that if we can verify an invariant on the abstracted specification — the specification with the equations defining the abstraction — then it also holds in the original specification. The implication, however, only works in one direction, if we find a counterexample in the abstracted system it does not necessarily mean that a counterexample exists for the original system.

Let us apply the technique to our example. There are several reasons why our Petri nets system is infinite. On the one hand, we have that tokens get accumulated in the Outside Station place, since every time a car gets its tank filled, the car leaves the station and a new token is added to the place. On the other hand, since we are representing tokens as objects, every time a new token is created it gets a new unique identifier. Thus, even though from the Petri nets point of view, tokens are anonymous dots in a place, in our representation tokens are objects that have names and identifiers, and the counter object keeps getting its value attribute increased. We may, however, abstract from this information, since neither the tokens' names nor identifiers are relevant, nor are we really concerned about the number of tokens we have in the Outside Station place. For the operation of the Petri net, the number of tokens in the other places is not relevant either. Specifically, since all arc weights are one, the analysis would be the same if having four or three tokens in the Station Tank place.

We introduce equations that abstract the Petri net system is the following way: (1) The Outside Station place gets its number of tokens decremented if it is bigger than one, (2) the number of tokens in the Station Tank place becomes 3 if it gets 4 tokens, (3) names and identifiers of token objects are reset into a range of values not used by the counter object, and (4) the counter object gets its value attribute restarted to its initial value. Notice that when we eliminate or rename a token, we also act on the relation object associating it to the place in which it is located. In this way, we not only make all places to have either zero or one tokens in them, but names and identifiers are reused from a small set of possible values.

This abstraction makes the state space finite. And we can use it to verify LTL formulas on the abstracted model. Temporal logic allows the specification of safety properties (something bad never happens) and liveness properties (something good eventually happens), which are related to the infinite behaviour of a system. However, we need a few additional definitions first.

Kripke structures are the natural models for propositional temporal logic. We need to understand how a Kripke structure is associated to the rewrite theory specified by a Maude system module. Basically, a Kripke structure is a (total) transition system to which we have added a collection of unary state predicates on its set of states. Therefore, since the models of rewrit-

ing logic are also transition systems, we need to make explicit the type of each of the states (System in our specification) and the atomic propositions on which we define our state predicates. In our case, again, we use OCL boolean expressions as basic propositions, associating to each state those boolean expressions that are satisfied in such a state. For example, we may check the following properties

- **Property 1.** To check the property stating that it is always true that eventually the system gets to a state in which there is a token on either the Disabled or the Outside Station place can be checked using the following LTL formula:

```
[] <> (id(4, "Disabled").tokens->size() > 0
       \/  id(4, "Outside Station").tokens->size() > 0)
```

  In this case, the answer obtained is positive.
- **Property 2.** The following LTL formula states that if we reach a state in which there is a token in the Fuel Indicator On place, then eventually a state in which there is a token in the Outside Station place is reached.

```
[] (id(4, "Fuel Indicator ON").tokens->size() > 0
   -> []<> id(4, "OutsideStation").tokens->size() > 0)
```

  In this case the response is negative. Indeed, it may happen that the Petri net loops in the upper part, evacuating and replenishing the station tank over and over again. As usual, if the formula is not true, the model checker gives a counterexample, in the form of a sequence of states.

The interested reader can find the complete outputs in [54], together with the source files used to reproduce the execution.

5 Related Work

Even though there exist a plethora of MLM approaches and tools, only a few of them support DSML behaviour specification and execution. Melanee [2] is one of the most advanced tools for MLM. The tool supports a variant of OCL with deep semantics (DeepOCL) which has been integrated with the Atlas Transformation Language (ATL) for model transformations. Lange shows in [31] how this tool can be used to check constraints spanning multiple classification levels which can be defined and executed. Although Melanee itself is not natively supporting tools for simulation/execution through the specification of the execution semantics, i.e., (multilevel) transformation rules, there are some works on top of it that aims to achieve this (see, e.g., [3]). In that work, the model execution mechanism is based on a service API and a plug-in mechanism, and the communication between the modelling and the execution environments is realised using socket-based communication. We provide in our approach the whole set of tools necessary to directly be able to define the structure of the multilevel hierarchy, specify the multilevel model transformation rules (MCMTs), execute/simulate the models, and analyse the system.

The MetaDepth tool [32] is a well-known framework within the MLM community. It is integrated with the Epsilon languages [20], which permits using the Epsilon Object Language (EOL) as an action language to define behaviour for metamodels, as well as the Epsilon Validation Language (EVL) for expressing constraints. Both EOL and EVL are extensions of OCL. The approach implements the interface of the connectivity layer in a way to make EOL aware of the multiple ontological levels providing it with a multilevel nature. However, the authors of [32] state that MetaDepth can be used as a normal two-level meta-modelling environment when it comes to the execution of behaviour of the models. Thus, for the actual execution they would have to flatten their multilevel language to a two-level version in order to run the models. To the best of our knowledge there is not yet a MLM tool that supports or integrates model checking and analysis capabilities within its MLM tool-set.

Other authors have attempted to handle pattern identification and specification to define reusable model transformation rules. There exist a diverse set of approaches that bring solutions to pattern definition and application in the context of graph transformations. In [24], Guerra and de Lara explore *recursion* as a graph transformation mechanism. They provide double pushout (DPO) rules with base and recursive conditions, together with mechanisms to pass the matching between successive recursion steps. Lindqvist et al. [36] propose the star operator, which is suited to find repetitive occurrences of a specific modelling pattern. However, the star operator is only defined for matching model extracts, and not to perform transformations. In [23], Grønmo, Krogdahl, and Møller-Pedersen present a collection operator for graph transformation and show its usage to a variety of Coloured Petri nets. Using this operator, it is possible to match several similar structures within the model. They theoretically define how nesting would work by producing an ad-hoc rule that would fit to the specific case. We follow a similar approach by defining the rule in a generic way and then the transformation engine provides also a generic version in the Maude specification. The advantage of our approach with respect to the collection operator is that we do not physically produce an unfolded rule, but

it is dynamically unfolded and used at run-time. It is during the matching at run-time when the rule is unfolded guided by the cardinalities provided in the rule. In [50], Rensink and Kuperus propose a transformation language that uses an amalgamation scheme for nested graph transformation rules, where pattern elements are combined with universal and existing quantifiers. The transformation language is used in the GROOVE tool. Henshin [60] is an in-place model transformation language for the EMF. Among other features, it implements a *rule-nesting* mechanism [1] that provides a for-each operator for rules. In nested rules, the outer rule is referred to as *kernel rule* and the inner rule as *multi rule*. During execution of a nested rule, the kernel rule is matched and executed once. Afterwards, the match is used as a starting point to match the multi-rule as often as possible and execute it for each match.

There are several traditional MDSE approaches that deal with execution and verification that are somehow related to our proposal. In [52], Rivera et at. use Maude to represent 2-level models to be able to simulate and perform formal analysis and model checking on them. Such work served us as inspiration and starting point. We were considering either implementing ourselves an execution engine within MultEcore or using an existing tool where to rely on. Studying some works where Maude was used and analysing how the language could be customised together with its plethora of existing capabilities made us to follow such a path. We also analysed other mature tools in the context of model execution via operational semantics, e.g., Henshin [60] or the GEMOC Studio [16, 10], which helped us to understand how the user could interact with the execution tools from an Eclipse-based application. In the context of verification, GROOVE (GRaph-based Object-Oriented VErification) [48, 22] is a tool for software model checking of object-oriented systems. It can be used for modelling, analysis and verification and integrates all these functionalities in an easy to use interactive GUI. While we already integrate into MultEcore some Maude functionalities for execution and verification, we still have to work in this direction to ease the process to the modeller. We see GROOVE as a good influence to achieve a better usability degree in MultEcore.

6 Conclusions and Future Work

In this paper have presented an infrastructure for execution and analysis of multilevel modelling languages.

To make this possible, we have integrated Maude into our tool MultEcore, making it possible not only to define our multilevel hierarchy (language) and specify the behaviour by means of MCMTs, but also carry on simulation and further model checking and analysis techniques. However, Maude is used as a backend tool, hidden to the user such that the interaction is entirely done with MultEcore, making the modeller unaware of the Maude details. Although in the last years several traditional two-level tools have provided support for the whole cycle of behavioural DMSLs (from design to simulation and verification), to the best of our knowledge this is the first work where a MLM tool incorporates capabilities to perform model checking and other formal analyses. We believe that the work presented in this paper can open new doors to the MLM field, as this tool can be used to define behavioural multilevel DSMLs, execute them, and verify them.

Apart from the major improvements in the infrastructure itself, we have improved and extended the MCMTs capabilities, by allowing nested boxes to represent collections, incorporated attribute manipulations, and specification of conditions. Basic support for OCL is also provided, which is very useful for the manipulation of attributes, the specification of box cardinalities, the specification of rule conditions, and the specification of expressions and conditions to be used in the formal checks, including LTL formulas.

To validate and demonstrate that our infrastructure works and that actual execution and analysis can be carried on, we have provided a case study where a Petri net model that captures a gas station is simulated applying consecutively the MCMT rules defined in MultEcore. The goal of this case study has been to evaluate the usability and practicability of the developed infrastructure. We are already considering how to evaluate our tool against other MLM approaches that allow the modeller to perform execution on models by defining in-place model transformations. We have validated and verified the modelled system using reachability analysis and model checking techniques on an abstracted version of the model. We refer the reader to the main MultEcore webpage [53] for further details and examples.

We plan to integrate into MCMTs the capabilities that a programming language brings such as reasoning about functions, expressions, type specifications, and data manipulations. Our current implementation of certain OCL functions represents a step towards this goal. We are already working on adapting the complete mOdCL (Maude + OCL) [19] to our Multilevel infrastructure so we can make use of the full power of OCL in a Multilevel context. This would allow us, for instance, to specify Coloured Petri nets [27] which combine classical Petri nets with a programming language [62].

While MCMTs are flexible with respect to further horizontal/vertical extensions, we identify a key point of improvement as being able to reuse META levels on

MCMTs into other rules. This would improve our approach with a higher degree of modularity and reusability. Ultimately, we plan to further advance on the interface that connects MultEcore to Maude, bringing more advanced functionalities such as an interactive editor, a smoother experience to the user with the graphical editor and additional Maude capabilities to, for instance, customise the strategy language and have more control on the execution and analysis.

References

1. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)., pp. 121–135 (2010). DOI 10.1007/978-3-642-16145-2_9
2. Atkinson, C., Gerbig, R.: Flexible Deep Modeling with Melanee. In: S. Betz, U. Reimer (eds.) Modellierung 2016, *LNI*, vol. 255, pp. 117–122. Gesellschaft für Informatik, Bonn (2016)
3. Atkinson, C., Gerbig, R., Metzger, N.: On the Execution of Deep Models. In: 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)., pp. 28–33 (2015)
4. Atkinson, C., Kühne, T.: Processes and products in a multilevel metamodeling architecture. International Journal of Software Engineering and Knowledge Engineering **11**(06), 761–783 (2001)
5. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 19–33 (2001). DOI 10.1007/3-540-45441-1_3
6. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling **7**(3), 345–359 (2008)
7. Atkinson, C., Kühne, T.: In defence of deep modelling. Inf. Softw. Technol. **64**, 36–51 (2015). DOI 10.1016/j.infsof.2015.03.010
8. Atkinson, C., Kühne, T.: On Evaluating Multi-level Modeling. In: Proceedings of MULTI @ MODELS, pp. 274–277 (2017)
9. Bernardinello, L., de Cindio, F.: A survey of basic net models and modular net classes. In: Advances in Petri Nets 1992, The DEMON Project, pp. 304–351. Springer (1992). DOI 10.1007/3-540-55610-9_177
10. Bousse, E., Wimmer, M.: Domain-level observation and control for compiled executable dsls. In: M. Kessentini, T. Yue, A. Pretschner, S. Voss, L. Burgueño (eds.) 22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019, Munich, Germany, September 15-20, 2019, pp. 150–160. IEEE (2019). DOI 10.1109/MODELS.2019.000-6
11. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers (2012). DOI 10.2200/S00441ED1V01Y201208SWE001
12. Cabot, J., Gogolla, M.: Object constraint language (OCL): A definitive guide. In: M. Bernardo, V. Cortellessa, A. Pierantonio (eds.) Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures, *Lecture Notes in Computer Science*, vol. 7320, pp. 58–90. Springer (2012). DOI 10.1007/978-3-642-30982-3_3
13. Clark, T., Warmer, J.: Object Modeling With the OCL: The Rationale Behind the Object Constraint Language, vol. 2263. Springer (2003)
14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285**(2), 187–243 (2002)
15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, *Lecture Notes in Computer Science*, vol. 4350. Springer (2007). DOI 10.1007/978-3-540-71999-1
16. Combemale, B., Barais, O., Wortmann, A.: Language engineering with the GEMOC studio. In: 2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017, pp. 189–191. IEEE Computer Society (2017). DOI 10.1109/ICSAW.2017.61
17. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Rubio, R., Talcott, C.L.: Programming and symbolic computation in Maude. J. Log. Algebraic Methods Program. **110** (2020). DOI 10.1016/j.jlamp.2019.100497. URL `https://doi.org/10.1016/j.jlamp.2019.100497`
18. Durán, F., Garavel, H.: The rewrite engines competitions: A rectrospective. In: D. Beyer, M. Huisman, F. Kordon, B. Steffen (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Proceedings, Part III, *Lecture Notes in Computer Science*, vol. 11429, pp. 93–100. Springer (2019). DOI 10.1007/978-3-030-17502-3_6. URL `https://doi.org/10.1007/978-3-030-17502-3_6`
19. Durán, F., Roldán, M.: Validating OCL constraints on Maude prototypes of UML models. Tech. rep., Universidad de Málaga (2012)
20. The Epsilon Object Language (EOL). https://www.eclipse.org/epsilon/doc/eol/
21. Garavel, H., Tabikh, M., Arrada, I.: Benchmarking implementations of term rewriting and pattern matching in algebraic, functional, and object-oriented languages - the 4th rewrite engines competition. In: V. Rusu (ed.) Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018, Held as a Satellite Event of ETAPS, Proceedings, *Lecture Notes in Computer Science*, vol. 11152, pp. 1–25. Springer (2018). DOI 10.1007/978-3-319-99840-4_1. URL `https://doi.org/10.1007/978-3-319-99840-4_1`
22. Ghamarian, A.H., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. Int. J. Softw. Tools Technol. Transf. **14**(1), 15–40 (2012). DOI 10.1007/s10009-011-0186-x
23. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. Software and Systems Modeling **12**(1), 121–144 (2013). DOI 10.1007/s10270-011-0190-3
24. Guerra, E., de Lara, J.: Adding Recursion to Graph Transformation. ECEASST **6** (2007). DOI 10.14279/tuj.eceasst.6.56
25. Halder, A., Venkateswarlu, A.: A study of petri nets modeling analysis and simulation. Department of Aerospace

Engineering Indian Institute of Technology Kharagpur, India (2006)
26. Hee, van, K., Leurs, M., Post, R.: Yasper : Yet another smart process editor (poster). In: 2005 Symposium on Verification and validation of software systems (VVSS 2005) (2005)
27. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer **9**(3), 213–254 (2007). DOI 10.1007/s10009-007-0038-x
28. Kelly, S., Tolvanen, J.: Domain-Specific Modeling - Enabling Full Code Generation. Wiley (2008)
29. Kühne, T.: Exploring Potency. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS, pp. 2–12 (2018). DOI 10.1145/3239372.3239411
30. Kühne, T.: A story of levels. In: Proceedings of MULTI @ MODELS, pp. 673–682 (2018)
31. Lange, A.: dACL: the deep constraint and action language for static and dynamic semantic definition in Melanee (2016). URL `https://madoc.bib.uni-mannheim.de/43490/`. Unpublished
32. de Lara, J., Guerra, E.: Deep meta-modelling with MetaDepth. In: Objects, Models, Components, Patterns, *lncs*, vol. 6141, pp. 1–20. springer (2010). DOI 10.1007/978-3-642-13953-6_1
33. de Lara, J., Guerra, E.: Generic Meta-modelling with Concepts, Templates and Mixin Layers. In: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS, pp. 16–30 (2010). DOI 10.1007/978-3-642-16145-2_2
34. de Lara, J., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Transactions on Software Engineering and Methodology (TOSEM) **24**(2), 12 (2014)
35. de Lara, J., Vangheluwe, H.: AToM 3: A Tool for Multi-formalism and Meta-modelling. In: International Conference on Fundamental Approaches to Software Engineering, pp. 174–188. Springer (2002)
36. Lindqvist, J., Lundkvist, T., Porres, I.: A Query Language With the Star Operator. ECEASST **6** (2007). DOI 10.14279/tuj.eceasst.6.55
37. Macías, F.: Multilevel modelling and domain-specific languages. PhD thesis, Western Norway University of Applied Sciences and University of Oslo (2019)
38. Macías, F., Rutle, A., Stolz, V.: Multilevel Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: Proceedings of MULTI @ MODELS, pp. 269–273 (2017)
39. Macías, F., Wolter, U., Rutle, A., Durán, F., Rodriguez-Echeverria, R.: Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour. Journal of Logical and Algebraic Methods in Programming **106**, 167–195 (2019). DOI 10.1016/j.jlamp.2018.12.005
40. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). DOI 10.1016/0304-3975(92)90182-F
41. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). DOI 10.1016/0304-3975(92)90182-F
42. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7-8), 721–781 (2012). DOI 10.1016/j.jlap.2012.06.003
43. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theor. Comput. Sci. **403**(2-3), 239–264 (2008). DOI 10.1016/j.tcs.2008.04.040. URL `https://doi.org/10.1016/j.tcs.2008.04.040`
44. Meta Object Facility (MOF) specification 2.5.1. https://www.omg.org/spec/MOF
45. Mohagheghi, P., Gilani, W., Stefanescu, A., Fernández, M.A., Nordmoen, B., Fritzsche, M.: Where does model-driven engineering help? Experiences from three industrial cases. Software & Systems Modeling **12**(3), 619–639 (2013)
46. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
47. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer (2013). DOI 10.1007/978-3-642-33278-4
48. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: J.L. Pfaltz, M. Nagl, B. Böhlen (eds.) Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, *Lecture Notes in Computer Science*, vol. 3062, pp. 479–485. Springer (2003). DOI 10.1007/978-3-540-25959-6_40
49. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: International Workshop on Applications of Graph Transformations with Industrial Relevance, pp. 479–485. Springer (2003)
50. Rensink, A., Kuperus, J.: Repotting the Geraniums: On Nested Graph Transformation Rules. Electronic Communication of the European Association of Software Science and Technology **18** (2009). DOI 10.14279/tuj.eceasst.18.260
51. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on, pp. 51–55. IEEE (2009)
52. Rivera, J.E., Durán, F., Vallecillo, A.: Formal Specification and Analysis of Domain Specific Models Using Maude. Simulation **85**(11-12), 778–792 (2009). DOI 10.1177/0037549709341635
53. Rodríguez, A., Durán, F., Kristensen, L.M.: MultEcore webpage (2021). URL `https://ict.hvl.no/multecore/`
54. Rodríguez, A., Durán, F., Kristensen, L.M.: Petri nets experiment resources: MultEcore and Maude files (2021). URL `https://bitbucket.org/phdalejandro/no.hvl.multecore.examples.sosym.petrinets`
55. Rodríguez, A., Durán, F., Rutle, A., Kristensen, L.M.: Executing Multilevel Domain-Specific Models in Maude. Journal of Object Technology **18**(2), 4:1–21 (2019). DOI 10.5381/jot.2019.18.2.a4
56. Rodríguez, A., Macías, F.: Multilevel Modelling with MultEcore: A Contribution to the MULTI Process Challenge. In: Proceedings of MULTI @ MODELS, pp. 152–163 (2019). DOI 10.1109/MODELS-C.2019.00026
57. Rodríguez, A., Rutle, A., Kristensen, L.M., Durán, F.: A Foundation for the Composition of Multilevel Domain-Specific Languages. In: MULTI@ MoDELS, pp. 88–97 (2019). DOI 10.1109/MODELS-C.2019.00018
58. Roldán, M., Durán, F.: Dynamic validation of OCL constraints with mOdCL. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **44** (2011). DOI 10.14279/tuj.eceasst.44.625
59. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
60. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In: 10th International Conference, ICGT 2017, pp. 196–208 (2017). DOI 10.1007/978-3-319-61470-0_12

61. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: AToMPM: A Web-based Modeling Environment. In: MODELS-JP 2013, *CEUR Workshop Proceedings*, vol. 1115, pp. 21–25 (2013)
62. Ullman, J.D.: Elements of ML programming. Prentice-Hall, Inc. (1994)
63. The Unified Modelling Language (UML) specification 2.5.1. https://www.omg.org/spec/UML
64. Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., Kühne, T.: Multi-level modelling in the Modelverse. In: MULTI@ MoDELS, *CEUR Workshop Proceedings*, vol. 1286, pp. 83–92 (2014)
65. Verbeek, H.M.W., Wynn, M.T., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Reduction rules for reset/inhibitor nets. J. Comput. Syst. Sci. **76**(2), 125–143 (2010). DOI 10.1016/j.jcss.2009.06.003
66. Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition: Getting Your Models Ready for MDA. Addison-Wesley Educational Publishers (2003)

A Petri nets multilevel hierarchy

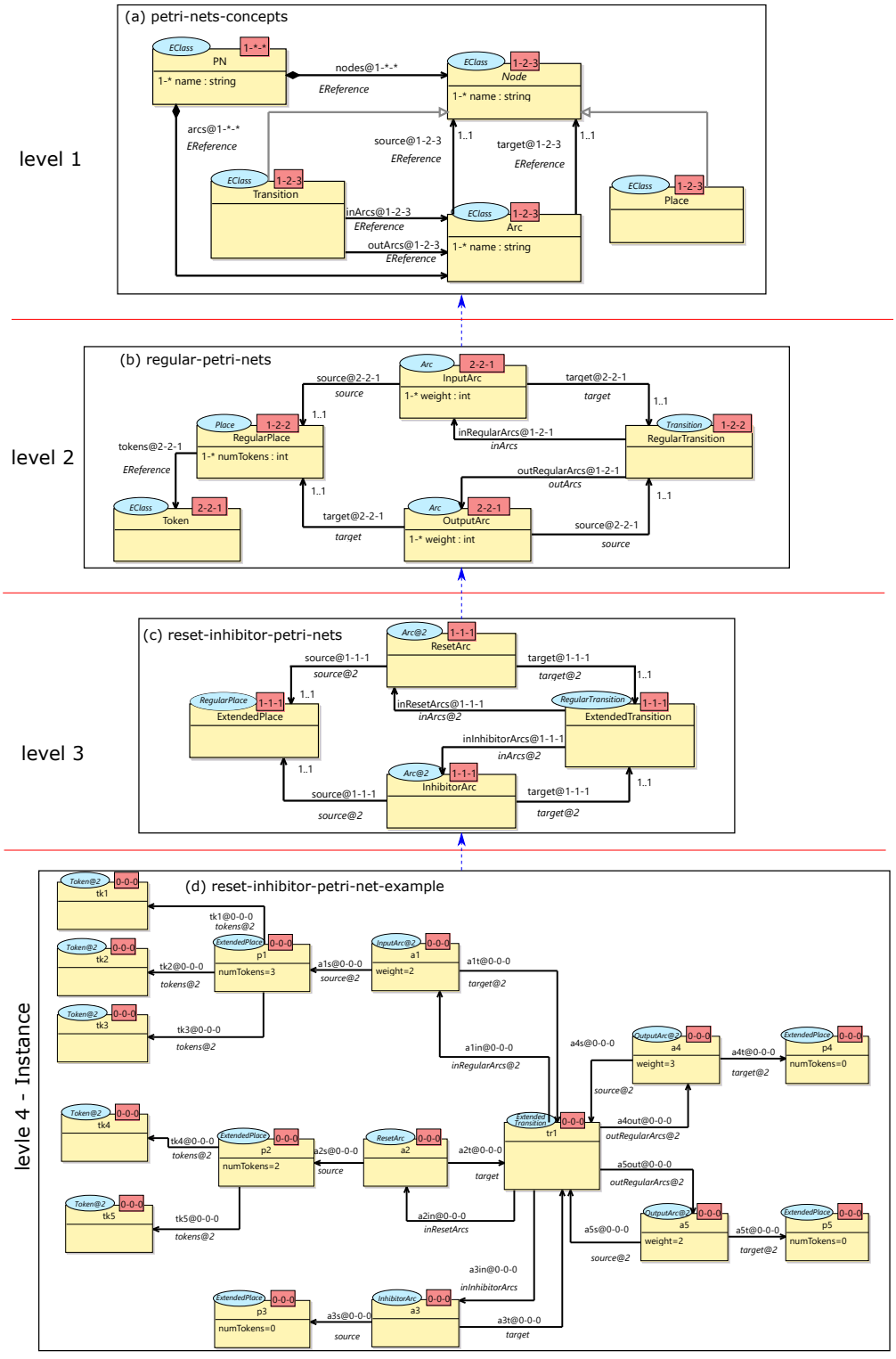

Fig. 19 Petri nets multilevel hierarchy