A Foundation for the Composition of Multilevel Domain-Specific Languages

Alejandro Rodríguez, Adrian Rutle, Lars Michael Kristensen
Department of Software Engineering
Western Norway University of Applied Sciences
Bergen, Norway

Francisco Durán
Department of Computer Science
Universidad de Málaga
Málaga, Spain

***Abstract*—In this paper, we provide a foundation for the definition and composition of multilevel domain-specific modelling languages. We will introduce modularization techniques such as composition, aggregation and referencing to enhance flexibility and reusability of these languages. To explain this foundation, we use Coloured Petri Nets (CPN) as a paradigmatic case study and define two CPN variants motivated by industrial collaboration projects: one used for the definition of protocols and the other one for robot controllers.**

***Index Terms*—Model-Driven Software Engineering, Model Transformations, Multilevel Modelling, Coloured Petri Nets**

I. Introduction

Model-driven software engineering (MDSE) it has proven to be a successful approach in terms of gaining quality and effectiveness [1]. It tackles the constantly growing complexity of software by utilizing abstractions and modelling techniques and considering models as first-class entities in all phases of software development. MDSE has also demonstrated to be an effective solution for the definition of industrial DSMLs [2].

However, traditional modelling approaches such as the Eclipse Modelling Framework (EMF) [3] and the Unified Modelling Language (UML) [4], have a limitation in the number of abstraction levels. Enforcing the designers to model systems within two levels can lead to several problems such as convolution and accidental complexity [5]. This challenge becomes more prevalent in the case of defining modelling languages that are tailored for a specific problem space — i.e., domain-specific modelling languages (DSMLs) — since variations of these languages would require further specializations of the metamodels. Multilevel modelling (MLM) tackles these problems by removing the limitation in the number of abstraction levels. Indeed, MLM has proven to be a successful approach in areas such as software architecture and process modelling domains [5], [6].

Even though DSMLs are aimed to capture specific domains and can be understood as unique, not all existing DSMLs are different among them. The research community in software language engineering has proposed the notion of *Language Product Lines Engineering* (LPLE) for the construction of software product lines where the products are languages [7]. The key of this approach is the definition of *language features* that encapsulate a set of language constructs representing certain DSMLs functionalities. Taking advantage of MLM and the concept of feature, we provide mechanisms to define the abstract syntax and the semantics in a modular way, i.e., one can add/remove new dimensions to a selected model or transformation rule consistently.

In this paper, we present a foundation for the definition and composition of multilevel DSMLs. We demonstrate the foundation using a multilevel hierarchy where two specializations for Coloured Petri Nets (CPN) [8] are defined. In this context, we exploit multilevel model transformations (i) to specify multilevel constraints (that check structural and semantical correctness of the model), i.e., static semantics, and (ii) to define model transformations for the definition of the execution behaviour of the models, i.e., dynamic semantics. Furthermore, we explain how we can reuse these model transformations, established for one branch, in other branches. Since the different languages are defined within the same family [9] (CPN in our case), reusability and flexibility are key, as the former facilitates sharing features between the different sub-languages, while the later makes adaptation easier. We also achieve the capability of composing DSMLs in a natural way, following the intuition of the *Trait* [10] and *Mixin* [11] concepts in object-oriented programming. Hence, this paper can be seen as a path to provide solutions to the challenges exposed in [7] for the reuse of similar DSLs by taking advantages among the commonalities between them.

We use our tool MultEcore to define both the structure and the semantics of the MLM hierarchy of CPN [12]. First, we design the structure using the graphical editor that allows the creation of multilevel hierarchies, then we define the behaviour by using the so-called Multilevel Coupled Model Transformations (MCMTs) [13], [14]. MCMTs are in this paper extended to specify both multilevel constraints and reusable CPN behaviour. Note that this way we can specify both static and dynamic semantics using the same constructs. For this, we take advantage of the improvements we have made to the so-called supplementary hierarchies, which incorporate new dimensions to the CPN hierarchy. This work was introduced in [15] where a supplementary hierarchy was defined to integrate data types to a multilevel hierarchy.

We transform MLM hierarchies and their MCMTs to Maude for execution and simulation purposes. Maude is a high-level language and a high-performance interpreter in the OBJ algebraic specification family [16]. This way, we ensure the correctness of the designed hierarchies and their behaviour. In [17] we describe the infrastructure that allows us to transform both the multilevel hierarchies and the MCMTs from MultEcore to Maude, perform the rewriting of our models applying the constraints/rules and then transform the results back to MultEcore. The semantics of the model executed in [17] is less complex than the one presented in this paper. Thus, we do not perform execution of models but we focus on describing the multilevel transformation rules and set the base for future work in this direction.

We decide to use MultEcore as some of its key features make it possible to construct the case shown in this paper. For instance, the use of supplementary hierarchies to incorporate aspects in our models that are not strictly related to the domain being modelled, or the MCMTs for specifying the semantics.

The paper is organised as follows. In Sect. II we introduce basic concepts and notations on CPN that are used along the rest of this paper. Sect. III describes the running example focusing on the structural aspects. In Sect. IV we complete the hierarchy shown in Sect. III by providing constraints (Sect. IV-A) and describing the behaviour (Sect. IV-B). In Sect. V we discuss related work. Finally, Sect. VI concludes the paper and outlines directions for future work.

II. Background

Coloured Petri Nets [18] belong to the family of high-level Petri nets [19], which are characterized for combining classical Petri nets [20] with a programming language [21]. The use of a programming language, Standard ML (SML) in CPN, provides the primitives for the definition of data types, for describing data manipulation and for creating compact and parameterisable models. One advantage of CPN is that they contain few but powerful modelling constructs. However, CPN is not designed to be easily extended with domain-specific features, despite several recent applications of CPN have shown that it would be beneficial to be able to develop domain-specific variants [22]. Furthermore, two ongoing industrial collaborative projects have only confirmed the need for domain-specific CPN variants.

Fig. 1 shows a small example of a CPN model. It can be understood as the client-side of an Internet of Things (IoT) like protocol. The model depicts a scenario where temperatures from external sensors would be read. These temperatures would then be processed, creating signals to, for example, turn a heater on or off if the temperature goes above or below certain thresholds, and finally send the signal out to the external actuator.

A CPN model describes the states (*places*, drawn as ellipses) of the system and the events (*transitions*, drawn as rectangles) that can cause the system to change its state. *Arcs* can connect places to transitions or vice versa, but it is illegal to connect places with places or transitions with transitions. We explain in Sect. IV-A how this kind of constraints can be specified using MCMTs. Each place has an associated *type* (also called *colour set* in CPN terminology) determining the kind of data that the place may contain. A place can hold an arbitrary number of *tokens* that provide the *marking* of the place. For instance, in Fig. 1, IncomingTemperature and OutSignal places have INT and STRING as types (corresponding to the primitive types *Integer* and *String*), respectively.

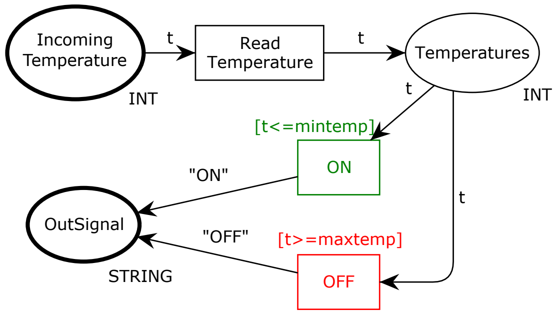

Fig. 1. CPN model for a temperature reader

Initial tokens in CPN models are produced by the so-called *initial marking* expressions. Specifically, the evaluation of the initial markings provides the tokens that the initial state of the model will contain. The state of a CPN is a marking of the places of the CPN model. The actions of a CPN consist of occurrences of enabled transitions. For a transition to be enabled it must be possible to find a binding of the variables that appear in the surrounding arc expressions of the transition such that the arc expression of each *input arc* evaluates to a multi-set of token colours that is present on the corresponding *input place*. The types of the arc expressions need to conform to the types of the places they are connected to. When a transition occurs with a given binding, it removes from each input place the multi-set of token colours to which the corresponding input arc expression evaluates. Analogously, it adds to each *output place* of the transition the multi-set of token colours to which the expression on the corresponding output arc evaluates. Such an occurrence changes the marking of the CPN model. Notice that the model in Fig. 1 does not show any token as we just want to show the structure (neither there is any initial marking specified). As mentioned above, the specific tokens added and removed by the occurrence of a transition are determined by the *arc expressions*, which are positioned next to the arcs (e.g., t and "ON").

In addition to the arc expressions, it is possible to attach a boolean expression to each transition. This boolean expression is used as a *guard* on transitions (e.g., [t>=maxtemp]). It specifies an additional requirement for the transition to be enabled (see [18], [23] for further details on CPN).

III. Multilevel Infrastructure for CPN DSMLs

MLM techniques match well with the creation of DSMLs, especially when we focus on behavioural languages since behaviour is usually defined at the metamodel level while it is executed at least two levels below (i.e., at the instance level) [24].

In order to facilitate the definition of domain-specific CPN variants, we have designed a MLM hierarchy to capture the structure and behaviour of a family of CPN models. The proposed infrastructure allows to incorporate/create new branches and modify the existing ones in a flexible way.

Figure 2 shows the complete CPN multilevel hierarchy using the MultEcore tool [14]. Figure 2a shows the generic_cpn metamodel which captures general purpose CPN. We can see the root node representing a Net in the top left corner of Fig. 2a. A net contains Nodes (which can be either a Place or a Transition), Arcs and Expressions. Expressions are used to represent CPN inscriptions (expressions) which in the current CPN Tools are defined using SML [25]. Also, a Place might hold several Tokens. A Transition can have several associated Bindings. The bindings are created by assigning Values to the free Variables of the transition (these free variables are detected by analysing the expressions of the arcs connected to the transition). The annotations in the rectangles at the top of the classes, and after the names in the arrows (separated by '@') specify the potencies.

Potency is used on elements as a means of restricting the levels at which this element may be used to type other elements. A potency specification includes three values: the first two specify the first and the last levels where one can directly instantiate an element (min and max), and the third value, the number of times the element can indirectly be re-instantiated (depth). Note that we specify the potencies of the elements as (1-*-*) since a concept might be instantiated several times. However, the concept Token should not be redefined and therefore the only instantiation could be done at the instance level. We do not show the instance level in Fig. 2 which would correspond to instantiations (snapshots) in a level below the protocol_config model represented in Fig. 2d. One can see (at the bottom right corner of Fig. 2a that the potency for such an element is expressed as (3-*-1), since we want to allow only one instantiation (depth = 1) of Token three (min = 3) or more (max = *) levels below—where '*' means unbound.

The next level on the hierarchy provide concepts for two different domains from the knowledge extracted by working with respective industrial partners. One could argue that such specializations can be carried on using inheritance in the same metamodel. However, this would lead to one bigger metamodel where specializations on elements (e.g., Place) that belong to different domains are put together and further extensions in each domain would be handled in such same metamodel. We rather introduce these levels to enhance modularization and separation of concerns, as MultEcore is designed to easily add such levels. The first one, called cpn_protocol (Fig. 2b), is aimed for CPN used to define protocols. For instance, QueuePlace is specified so it would incorporate the notion of Queue. Note that tokens in regular CPN places can be removed no matter the order they arrived to a place. In several protocols, however, order is a major feature that needs to be preserved. A ResetArc that connects a place with a transition empties all the tokens of the place whenever the transition is fired. This is useful as a "cleaning mechanism" in models that capture certain environments where messages might be retransmitted and buffers could accumulate old messages. Furthermore, InhibitorArc 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 it, not its presence. For instance, they can be used to delay certain actions until a system is idle, or to wait until the end of a loop.

The second CPN variant is cpn_controller (Fig. 2c) where InPlace (input) and OutPlace (output) are aimed to share information (introduce, or send, respectively) from the model with an external resource. For instance, InPlace will be used to introduce information from the outside environment to the model, and therefore it cannot be the target of any arc as it cannot receive tokens coming from the model. Analogously, OutPlace can only push information out, and it cannot be the source of an arc in the model. In CPN, arc expressions are built from typed variables, constants, operators, and functions. However, a requirement for an arc connected to an InPlace or an OutPlace is that the expression of such an arc can only be a variable that carries the information shared with the external environment.

The model defined in Fig. 2d represents a specific configuration for the protocol domain (protocol_config). We do not show a controller configuration for the lack of space, but it could be defined likewise. Models in the lowest level contain specific instances of the concepts defined in the levels above and they are used to specify concrete CPN configurations. Note that in our example, the *controller* branch is used for specifying constraints on the *protocol* hierarchy. One can see this as an application of *Referencing* (see [7] for details).

The CPN hierarchy we describe (Fig. 2) is called *application hierarchy*, and it can be understood as the main hierarchy. Application hierarchies can optionally include an arbitrary number of *supplementary hierarchies*, which add new dimensions to the application one. Using MultEcore, model designers can work with different multilevel hierarchies and fuse them with each other. This allows the concepts to have at least one type from the levels above in the application hierarchy and potentially one other type per incorporated supplementary hierarchy [12], [13]. This structure enhances modularization. The facility of adding/removing supplementary hierarchies is aligned with the *Aggregation* technique for language modularization [7] which provides a strong separation of concerns and strengthen reusability.

In our case, we consider the protocol branch as the application hierarchy which has three supplementary hierarchies:

Data types. A key difference of CPN (with respect to traditional Petri nets) is that one can define data types for the models. Data type declarations might include primitive data types (String, Integer, ...) or user-defined data types.

Error. One of the novelties we are exploring is the possibility to specify constraints that would raise errors when construct-

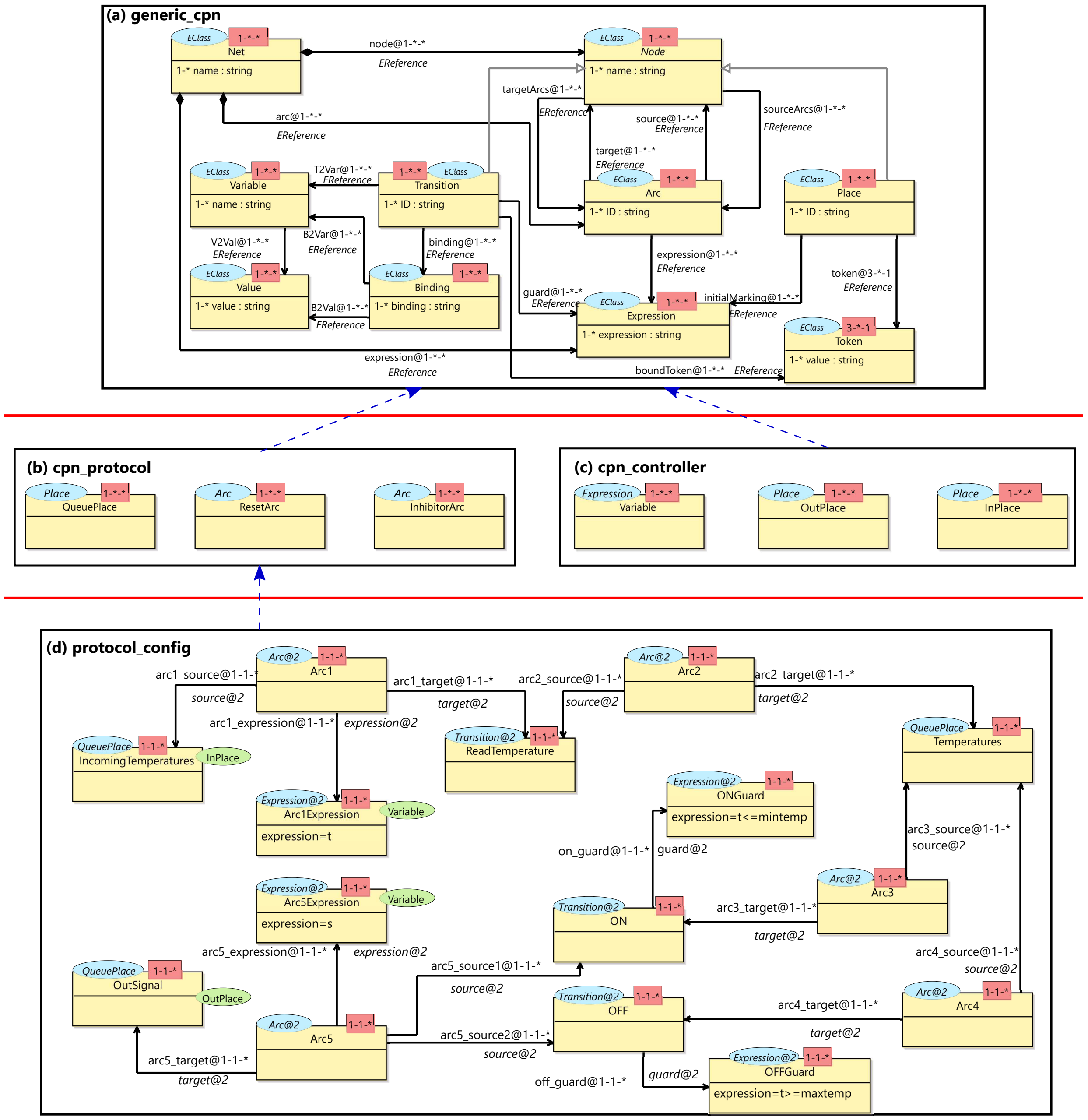

Fig. 2. CPN multilevel hierarchy

ing the models. Giving the elements of the model an extra supplementary type for errors would allow us to “tag” such elements when some constraint is violated.

Controller. We are also exploring a way to reuse concepts (and their semantics) from other branches; we have permitted to use the controller branch as a supplementary hierarchy, as the semantics of some concepts defined in cpn_controller will be useful for the cpn_protocol model shown in Fig. 2d.

We neither show *Data types* nor *Error* supplementary hierarchies (*Error* would only consist of one model with one class since this hierarchy is used to tag elements violating the constraints, and for this purpose it is sufficient to use such a simple supplementary hierarchy). However, we present in Fig. 3 a sketch of the situation where the application hierarchy incorporates supplementary ones. For the example, we have chosen the OutSignal element from the Protocol hierarchy. The type of this element, which comes from the application hierarchy, is defined in the level above. It also has OutPlace as supplementary type from the Controller hierarchy (right-side in Fig. 3) and String from the Data types hierarchy (left-side in Fig. 3). Notice that supplementary types can be used across different models (and levels) in the application one (for instance, in the SNAPSHOT level, tokens will be carrying supplementary data types as well).

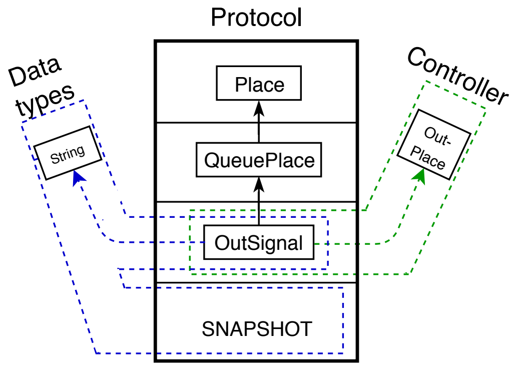

Fig. 3. Application hierarchy incorporating supplementary ones

The model cpn_protocol shown in Fig. 2d models the same behaviour as the CPN model shown in Fig. 1. Now we describe the model and highlight the key points where it gets benefits from the possibilities of extending the CPN concepts. IncomingTemperatures (top left of Fig. 2d) receives temperature values from an external source. One can see that this element is of type QueuePlace defined one level above in cpn_protocol (Fig. 2b) and it also has as supplementary type InPlace (expressed in cpn_controller, Fig. 2c). Having such types will give us the possibility of applying the semantics defined for both QueuePlace and InPlace. As explained above, one of the requirements for InPlace is that arc expressions of arcs going out of such a place can only be variables. This can be depicted in Arc1Expression where the supplementary type is Variable. The temperature will be then transferred to Temperatures which is of type QueuePlace (to preserve the order of the different temperatures received). Temperatures is connected, via Arc3 and Arc4, to two transitions, ON and OFF, respectively (bottom right of Fig. 2d). In this part of the model, a signal to turn a heater on will be sent if the temperature read is below the minimum temperature of the system (expression=t<=mintemp in ONGuard element).

Analogously, a signal to turn the heater off will be sent if the temperature is above the maximum temperature (expression=t>=maxtemp in the OFFGuard element). Finally, the OutSignal element will send out the signal generated by the model. The supplementary type OutPlace provides the additional behaviour (already explained) to such an element. Note that protocol_config (Fig. 2d) represents the structure of the modelled environment. An executable model would consist of a snapshot of protocol_config (instantiated in a level below) which is obtained by evaluating the initial marking expressions that produce the initial tokens.

IV. Multilevel Coupled Model Transformations

We have shown a multilevel configuration composed by two CPN variants. This section describes how we define the semantics for these models. Note that semantics of CPN are defined by a combination of model transformations and the functionalities of an orthogonal programming language which corresponds to the use of SML in CPN. The programming language takes care of the evaluation of expressions and type-correctness. In the following, we focus only on the model transformations part and hint the semantics that the programming language should accomplish for a complete specification.

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

MCMTs have been proposed as a mean to overcome the issues of both the traditional two-level transformations rules and the multilevel model transformations. While the former lacks the ability to capture generalities, the later is too loose to be precise enough (case distinctions) [14]. In this section we describe two applications of MCMTs. First, and as a novelty, for defining multilevel constraints, and second, to specify the behaviour of a multilevel DSML. Both of them reflect the new improvements and extensions we have incorporated to the MCMTs version described in [14]. Such improvements allow us to reason about the semantics that a complex modelling language such a CPN demands.

A. Multilevel constraints

Constraints are commonly specified using a constraint language, such as the Object Constraint Language (OCL) [26]. While these languages fit perfectly for defining constraints when they are applied in traditional two-level approaches, they might not be so appropriate for a multilevel setting. One could argue that defining OCL constraints on the superclass would solve the problem. However, this would imply to flatten the multilevel hierarchy into one level leading to concerns that MLM aims to solve. Even though there exists approaches that explore deep constraint languages for multilevel settings [27], the rule must refer to which level(s) it will affect (*scope*). With MCMTs, we avoid having to refer explicitly to the levels so, for instance, vertical extensions on the hierarchy would not affect the rules.

One of the capabilities of MCMTs is the possibility to specify multilevel constraints to check the semantical correctness of our models. Note that the constraints we describe in this section are formulated as transformation rules. In this section, we outline the usage of MCMTs for the definition of 3 multilevel constraints of which 2 are applicable to all CPN language variants and 1 is specific for a branch in the MLM hierarchy.

Illegal Arcs: Fig. 4 shows a constraint called *IllegalArc* that is triggered when an arc between two places is found. The META block allows us to locate types in any level of the

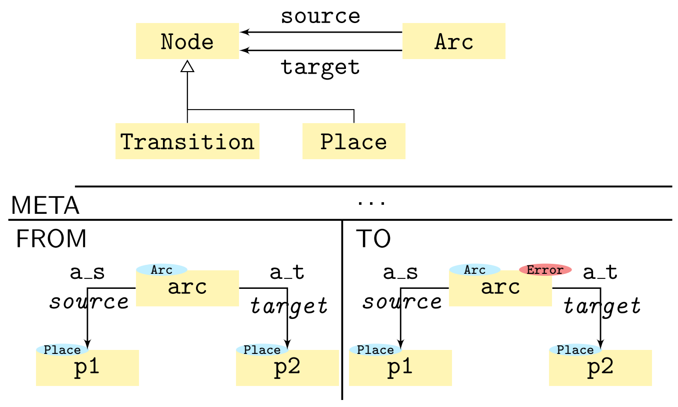

Fig. 4. Constraint *IllegalArc*: An error is triggered when an arc between two places is found

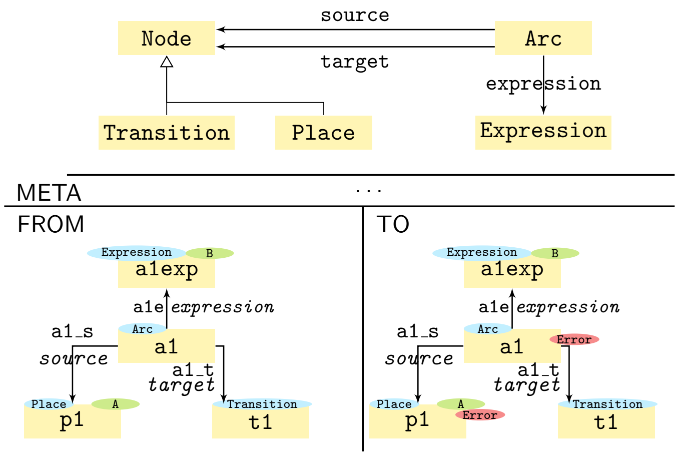

Fig. 5. *IllegalType*: Illegal typing between arcs and connected places

hierarchy that can be used in FROM and TO blocks. But the actual power of the META comes from the fact that we can use it to define an entire multilevel pattern. At the top level, we mirror part of generic_cpns (Fig. 2a), defining elements like Node, Arc, Place and Transition as nodes, and source and target as edges, that can be used directly as types in the FROM and TO blocks. One of the extensions on MCMTs we have made is the possibility to handle inheritance relations in the rules, e.g. Place and Transition inherit from Node. In this case, Place and Transition can be connected to Arc via a source or a target edge. These elements are defined as constants, meaning that the name of the pattern element must match an element with the same name in the MLM hierarchy on which we apply the rule.

A match of this rule is when the elements of the MLM hierarchy coupled together with their types fit two instances of Place which have relations of type source and target to an instance of Arc. Hence, if in the model we find a pattern where a place p1 is connected to another place p2 with an arc (FROM block) then we apply the changes specified in the TO block. In this case, we take advantage of the supplementary dimension adding the type Error to arc. This is just a way to notify the modeller that there is a mistake in the model without performing any reparation; it is up to the modeller to make the correcting changes.

The levels specified in an MCMT rule do not need to be consecutive, providing a more flexible definition. There might be several levels in between the blocks FROM/TO and the upper level. This is represented by the three dots in Fig. 4. This means that this rule can also be applied for every variant of the CPN language we are working with (i.e. the left or right branches shown in Fig. 2). Moreover, even if we add new levels in between, the rule would still be applicable. This is possible because we allow the types of the elements in FROM/TO to be *transitive* (i.e., indirect typing). For instance, in the FROM block of Fig. 4, the element p1 will match any node which directly or indirectly has Place as type. Whether p1 has as type Place or, for example, OutPlace, there will be a match and therefore an error will be detected indicating that it is not possible to have an arc between two elements of the type Place/OutPlace. As mentioned in Sect. II, it is also illegal to have arcs between two transitions. We need to define another constraint for specifying this constraint, however, we do not show this rule since it is essentially similar to the one shown in Fig. 4; the only difference is that variables p1 and p2 (they might be called t1 and t2) would be typed by Transition instead of Place.

In the example above, we just need to specify the constraint one time and it is applicable, not only to instances of places and transitions, but also for instances of the specializations of such elements made in levels below. This reusability would not be possible, for instance, with OCL, where we would need to write specific constraints for the elements defined in models below. E.g, for both QueuePlace in Fig. 2b or OutPlace in Fig. 2c, we would need to specify rules that disallow the creation of arcs between two of any of those elements.

Illegal Type: Another constraint which applies to all CPN language variants is that the type returned from the evaluation of an arc expression needs to conform to the type of the place where it is connected. For instance, it is illegal to have a place with type *String* connected to an arc that evaluates to *Integer*. *IllegalType* constraint shown in Fig. 5 checks the aforementioned property. Precisely, this rule captures illegal types between input places and their correspondent arc expression on arcs connected to them. The ellipses in the right top corner present in p1 and a1exp are aimed to express their types (from the *data types* supplementary hierarchy). Since A and B are different variables and we can bind them to different types, the match would success returning the mistake detected in the model. If so, error tags (from the *Error* supplementary hierarchy) may be added to p1 and a1 elements (TO block). We omit the rule for output places due to similarity.

Illegal Outplace: The third constraint *IllegalOutPlace* forbids the possibility of having an arc that has an instance of OutPlace as source (see Fig. 6). Recall that in the controller domain, out-places are used to share information with the external environment, and therefore they cannot be used to pass information inside the model. The figure shows two levels specified in the META block, separated by the red double-line.

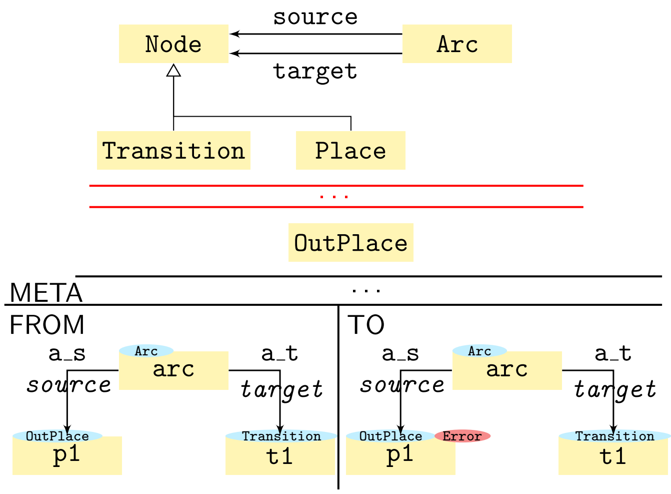

Fig. 6. *IllegalOutPlace*: error is triggered when OutPlace is source of an arc

Like the constraints already explained, the three dots "..." between the specified meta levels enhance the flexibility of the rule making it applicable in several cases without modifying it. Also, it leads to a more natural way of defining that a type is defined at some level above, without the need of saying explicitly in which level. The concept OutPlace defined in the second META level is a constant like the concepts defined in the top META. When an arc has as source (a_s) a place p1 of type OutPlace and we find a transition t1 as target (a_t), the place p1 is updated with the extra type Error notifying that there is a mistake in it.

A similar constraint is used to detect whether an InPlace is the target of some arc. InPlaces (and OutPlaces) can be understood as some sort of interfaces that are input (and output) points in the model which are connected to external resources. Thus, it is not possible to have transitions that produce tokens in InPlaces. However, we do not show such a constraint since it is essentially the same as Fig. 6. The only differences would be in the type of p1 (which should be InPlace) and in the types of the edges that connects arc with p1 and arc with t1 (they should be target and source, respectively). Also InPlace shall be declared in the second META level instead of OutPlace.

B. MCMTs for defining behaviour

MCMTs can be also used for describing the behaviour of a system. Specifying multilevel transformation rules that allow the execution of a model with MCMTs follows the same logic as the one presented in Sect. IV-A. However, they belong to two different stages. While the constraints are checked during the modelling process to analyse structural and semantical correctness, rules that describe the behaviour will be executed once the model has been initialized and instantiated one level below. In this section we display the behavioural rules that describe how CPN models are executed. An execution of a CPN model is, in general, described by means of an *occurrence sequence*.

As mentioned in Sect. III, we describe the behaviour of CPN models by model transformations together with a programming language which is used to define and evaluate the expressions and guards. First of all, we indicate three functions that should be implemented by the chosen programming language. $Var(e)$ is a function that takes an expression e and returns the variables used in such an expression. $Eval(e, b)$ evaluates an expression e over a binding b, and provides the multiset of tokens to be removed (if it is an input arc) or added (if it is an output arc) from the place connected to the arc. $Type(e)$ returns the type of an expression e.

The rules perform a step from one state to the next one during the simulation of a model. Such a step is performed when an enabled transition is fired. Recall that, for a transition to be enabled, it must be possible to find a binding of the variables that appear in the surrounding arc expressions that evaluates to a multi-set of token colours that is present on each corresponding input place. Then, the occurrence of a transition with a given binding removes from each input place tokens to which the corresponding input arc expression evaluates. Finally, it adds to each output place the tokens to which the expression on the corresponding output arc evaluates.

It is important to mention that there is not a direct relation between the tokens residing in a place and the binding of the transition connected to such a place. First, a transition can have assigned an arbitrary number of free variables that come from all the variables used in the arc expressions connected to the transition. Second, the bindings of the transitions are calculated assigning values to those free variables of the connected arcs (both input and output arcs). The bindings are static and do not change as long as the arc expressions are not modified. Thus, we assume there is a step before being able to simulate the model, where all the bindings of the transitions in the model are previously computed (for this, it will be necessary to use the function $Var(e)$ for every expression e of each arc connected to the transition).

A first rule called *EnableTransition* and shown in Fig. 7 creates the relation between the tokens that might be consumed from a place determined by the evaluation of the binding with the arc expression — this is provided by the function $Eval(e, b)$. It also creates the new tokens that can potentially be added to the output places connected to the transition. Note that these new tokens are not connected to output places, as the transition has not been fired yet. Variables [M] and [N] in the META level on the sourceArcs and token relations, represent the cardinality that have to be matched in order to apply the rule. An arbitrary number of places can be connected to a transition via input arcs. All the expressions of those input arcs have impact on selecting the tokens that might be removed from the connected places. Analogously, all the expressions of the output arcs have impact on producing the new tokens in the output places. We do not show the default multiplicities in Fig. 2a which is 0..* for sourceArcs and token relations. This means that [M] and [N] will be bounded to *, expressing that the match needs to be done with the biggest set of elements comprised in the dashed boxes that

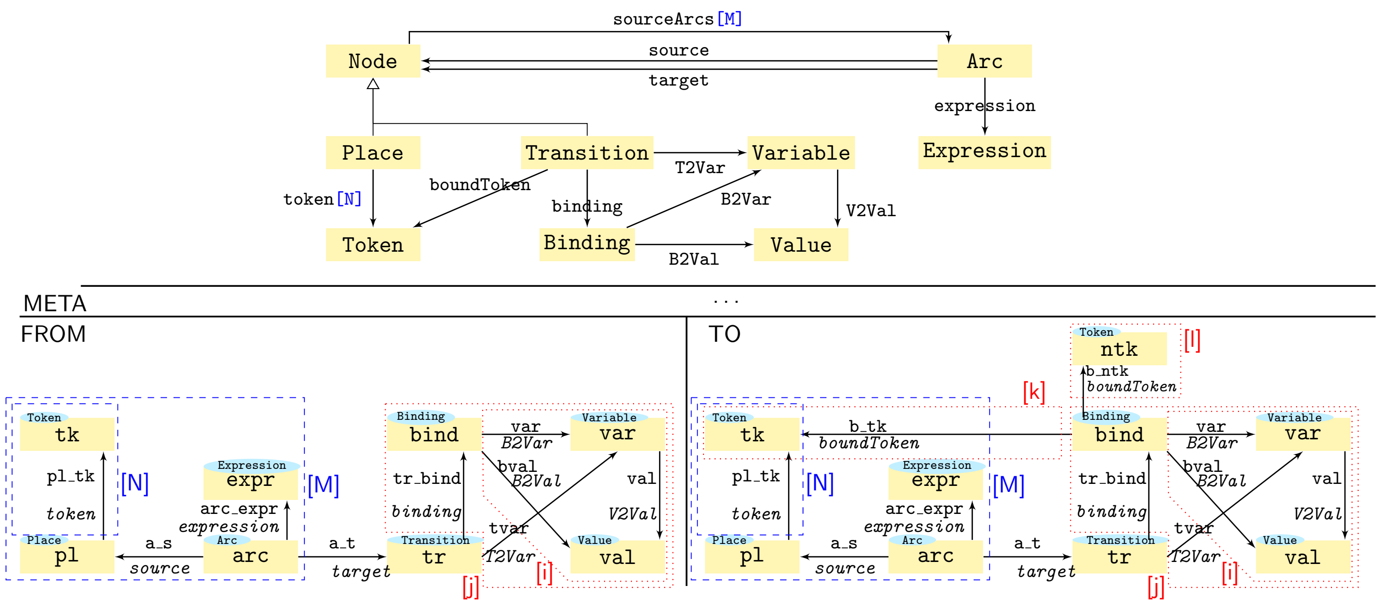

Fig. 7. Rule *EnableTransition*: enables a transition through evaluating the arc expressions with the bindings

fulfil the pattern defined in the FROM/TO blocks. When the multiplicity has been explicitly specified in the META level of the rule and we have instantiated those same variables for the multiplicities in the FROM block, then the rule will take that into consideration during the matching process. In the case of M, a successful match will occur when all the places (pl) that are connected to a transition tr via an arc are found. Having tr (in FROM block) out of the biggest dashed box makes sure that there is only one transition to which all the tuples arc-arcxpr-pl-token are connected. Note that a place can also hold an arbitrary number of tokens that might be affected by the evaluation of the binding against the correspondent arc expression. We therefore wrap the relation pl_tk and the node tk in another box bounded with the variable N. The way we define and use these multiplicities is inspired by the concept of *cardinality* described in [28].

The execution of this rule (FROM $\rightarrow$ TO) defines which tokens can be removed from the input places connected to the transition tr (it also defines which tokens can be produced in the output places).

In this step, for each j binding (bind), the function $Eval(expr, bind)$ is called taking each arc expression expr and the bind to select the tokens that can be removed from each place (if there are enough). Note that the biggest dotted shape (with the cardinality j) captures that a transition might have several bindings. The dotted box inside the biggest one (scoped with the cardinality i) expresses that a binding is composed of i variables var with each one having a value val assigned (see bottom right of Fig. 7). We use a different notation for the boxes as they are conceptually different. Even though they both express replication of the elements inside them, the dashed boxes are aimed to express the cardinality for the amount of elements to be found in the model, while the dotted boxes represent information that will be handled by the programming language. Notice in the TO block there is a new dotted box that wraps b_tk edge and tk node. This displays that bind has associated k tokens that can be removed on the occurrence of the transition. The dotted box that bundles b_ntk and ntk is, in a similar way than the previous box, capturing the new tokens that come from the evaluation of bind against the expression of each output arc (for simplicity, we do not show neither the output arcs nor the output places in the rule). As mentioned before, these new tokens have not really been produced yet, as this is performed when a transition is fired, therefore there are not yet relations between the new tokens (ntk) and the output places.

Recall that a transition is enabled if the required tokens are present on input places connected to input arcs of the transition. An occurrence of a transition removes tokens from input places, and adds tokens to output places connected to output arcs of the transition. We show in Fig. 8 the rule *Occurrence* of an enabled transition. The FROM block depicts the situation after the execution of the *EnableTransition* rule (Fig. 7). Again, M, N and O in the META block are variables to be matched with the multiplicities in the CPN model. On one hand, for each input place p1 connected with an arc a1 to a transition tr (captured by the dashed shape with the variable M), we select the subset of N tokens tk that reside in p1 and have been picked by applying $Eval()$ function for each arc expression with the selected bind in tr (this was already done in *EnableTransition* rule). Each k selected token (dotted rectangle wrapping b_tk and tk) has an edge b_tk between bind and tk. On the other hand, we have each output place p2 connected with an arc a2 to tr (defined by the dashed shape with the O variable). Recall that the execution of *EnableTransition* rule created the potential tokens that would be added in the output places (resulting from the evaluation of each output arc expression with the binding of tr) in case the transition were fired. These l new tokens are represented by ntk connected to bind via the relation b_ntk, and they are wrapped by the dashed box with l cardinality.

As the result of executing the *Occurrence* rule, (in TO block), the chosen tokens tk are removed from each input place p1, and the new tokens ntk are added to each output place p2. Note that the binding is not removed (as it does not change during the simulation) but it is cleaned so it can be used again in the next state of the model.

A third rule called *DisableTransition* might be specified for disabling transitions. This rule is similar to *EnableTransition* but it has the opposite effect. We do not show such a rule but state its intuition. When a transition is fired, it changes the marking of the model and some places that had tokens before might be empty after the occurrence. It is necessary to re-evaluate (disabling and enabling again) the model and calculate again which transitions can be enabled.

The constraints and behavioural rules presented in this section perform the first step towards the simulation of multilevel CPN models. The current state of MultEcore relies on a bidirectional transformation to Maude, in which the models are executed and their states are returned to the modelling tool. As discussed Sect. III, one of the key points of our approach is the reusability and flexibility of the transformation rules. Although the rules explained in this section specify the behaviour of general CPN models, they can be reused for various CPN variants (differing from the *protocol* and *controller* examples) since they are defined at a higher level of abstraction in the hierarchy (see [17] for details).

V. Related work

In this section, we present other approaches to the definition of infrastructures for modelling and executing behavioural DSMLs. In [29], [30], the authors present transformation product lines. This approach facilitates systematic creation of transformations for language families. Similarly, authors in [31] propose featured model transformations (FMTs) which can bee seen as a kind of metamodel that integrates the variability of a whole family of metamodels. Our approach considers not only the reusability of the transformation rules within the same family, but also allows the incorporation of orthogonal languages. In [27], the authors present an approach to transform a multilevel hierarchy to a two-level configuration (and vice versa) by adapting the ATL transformation language [32] making it "multi-level aware". Our approach goes one step further and facilitates the definition of multilevel model transformations and the behaviour of multilevel DSMLs by using MCMTs.

In the Language Product Lines Engineering field [7], Melange [33] is a tool that supports the construction of DSLs that supports modular language design and language modules composition. Operational semantics of a DSL involves the use of an action language to define methods that are statically introduced directly in the concepts of the DSL abstract syntax. In our approach, we define the semantics separately, by means of MCMTs, avoiding the need to change the abstract syntax (i.e. the multilevel hierarchy) of the DSML. MontiCore [34] supports the construction of textual DSLs where the abstract syntax is defined in BNF-like grammars. Languages can extend each other and can be embedded within each other.

We find some relevant work in the context of modular modelling. Reuseware [35] is a metamodel-agnostic approach to aspect-oriented modelling (AOM). The process for the composition requires the designer to manually define the *addressable points* which are used to specify either fragments of a model or points to be replaced by a fragment from some other model. In our approach we aim to be as less invasive as possible during the composition process.

VI. Conclusions and future work

In this paper, we have described an infrastructure for the definition and composition of multilevel DSMLs. We have proposed MCMTs to describe both (i) multilevel constraints (that check the structural and semantical correctness of models) and (ii) the reusable and flexible description of the behaviour of multilevel models. Moreover, we have used supplementary dimensions where new additional types can be added to elements in an MLM hierarchy. A key advantage that comes from the use of supplementary hierarchies (and therefore, the use of more than one dimension) is the combination of DSMLs to build more complete models that can be executed. This opens up for flexibility and reusability when defining related languages tailored for specific domains.

We have established a foundation for the use of MLM composition to describe behaviour based on the semantics of more than one DSML — in a way similar to what was done for two-level DSMLs in [36]. We believe that further investigation in this direction will lead us to the application of modularization and composition techniques, as detailed in [7], to exploit reusability in the development of DSMLs. Although we have outlined in the model transformations for enabling and firing transitions, the choice of the programming language for the evaluation of arc expressions, type checking or token generation is left open for future work.

References

[1] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2014.

[2] P. Mohagheghi, W. Gilani, A. Stefanescu, M. A. Fernández, B. Nordmoen, and M. Fritzsche, "Where does model-driven engineering help? Experiences from three industrial cases," *Software & Systems Modeling*, vol. 12, no. 3, pp. 619–639, 2013.

[3] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[4] "UML," http://www.uml.org/.

[5] J. D. Lara, E. Guerra, and J. S. Cuadrado, "When and how to use multilevel modelling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 12, 2014.

[6] C. Atkinson and T. Kühne, "On evaluating multi-level modeling," in *MODELS*, 2017.

[7] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry, "Leveraging software product lines engineering in the development of external dsls: A systematic literature review," *Computer Languages, Systems & Structures*, vol. 46, pp. 206–235, 2016.

[8] K. Jensen and L. M. Kristensen, *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media, 2009.

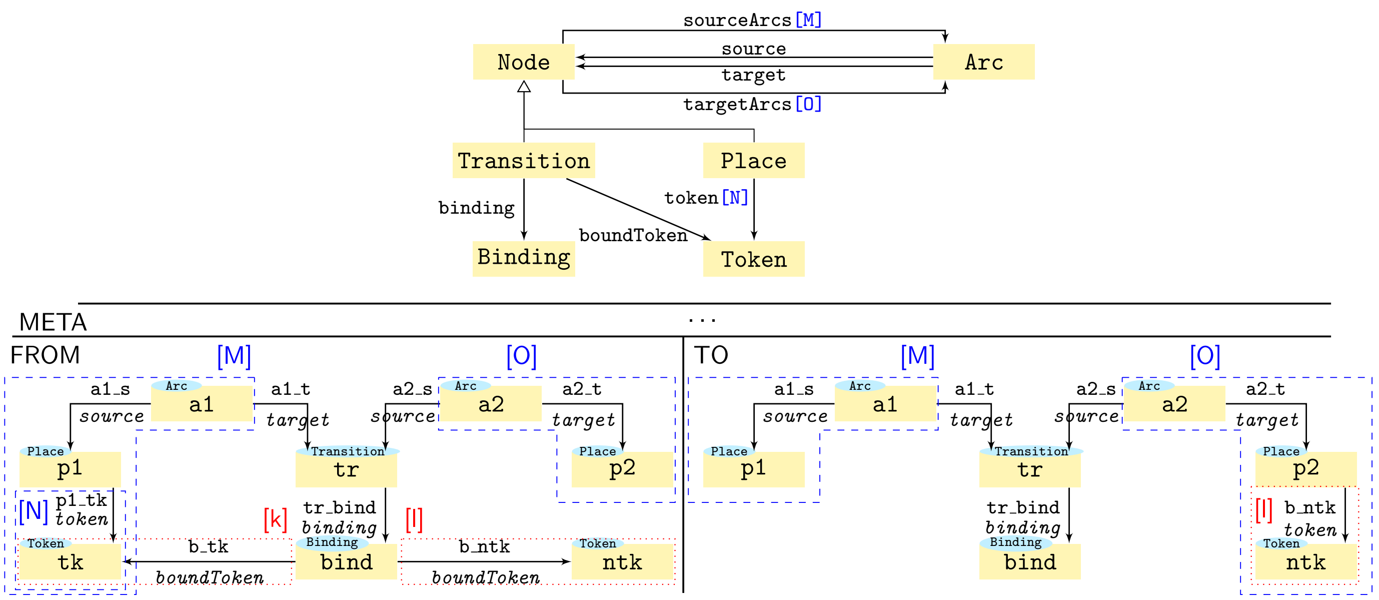

Fig. 8. Rule *Occurrence*: It fires a transition (it removes tokens from input places and creates tokens in output places)

[9] M. Voelter, “A family of languages for architecture description,” in *8th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, 2008, pp. 86–93.

[10] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, “Traits: Composable units of behaviour,” in *European Conference on Object-Oriented Programming*. Springer, 2003, pp. 248–274.

[11] G. Bracha and W. R. Cook, “Mixin-based inheritance,” in *OOPSLA/ECOOP, Ottawa, Canada, October 21-25, 1990, Proceedings.*, 1990, pp. 303–311.

[12] F. Macías, A. Rutle, and V. Stolz, “Multecore: Combining the best of fixed-level and multilevel metamodelling.” in *MULTI@ MoDELS*, 2016, pp. 66–75.

[13] F. Macías, A. Rutle, V. Stolz, R. Rodriguez-Echeverria, and U. Wolter, “An approach to flexible multilevel modelling,” *Enterprise Modelling and Information Systems Architectures*, vol. 13, pp. 10:1–10:35, 2018.

[14] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria, “Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour,” *Journal of Logical and Algebraic Methods in Programming*, 2019.

[15] A. Rodríguez, A. Rutle, F. Durán, L. M. Kristensen, and F. Macías, “Multilevel modelling of coloured petri nets,” in *MULTI@ MoDELS*, 2018, pp. 663–672.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about Maude a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[17] A. Rodríguez, F. Durán, A. Rutle, and L. M. Kristensen, “Executing Multilevel Domain-Specific Models in Maude,” *Journal of Object Technology*, vol. 18, no. 2, pp. 4:1–21, 2019.

[18] K. Jensen, L. M. Kristensen, and L. Wells, “Coloured petri nets and cpn tools for modelling and validation of concurrent systems,” *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 3, pp. 213–254, Jun 2007.

[19] K. Jensen and G. Rozenberg, *High-level Petri nets: theory and application*. Springer Science & Business Media, 2012.

[20] W. Reisig, *Petri nets: an introduction*. Springer Science & Business Media, 2012, vol. 4.

[21] J. D. Ullman, *Elements of ML programming*. Prentice-Hall, Inc., 1994.

[22] K. I. F. Simonsen, L. M. Kristensen, and E. Kindler, “Pragmatics annotated coloured petri nets for protocol software generation and verification,” in *Transactions on Petri Nets and Other Models of Concurrency XI*. Springer, 2016, pp. 1–27.

[23] L. M. Kristensen and S. Christensen, “Implementing coloured petri nets using a functional programming language,” *Higher-order and symbolic computation*, vol. 17, no. 3, pp. 207–243, 2004.

[24] A. Rutle, F. Macías, F. Durán, R. Rodriguez-Echeverría, and U. Wolter, “Describing Behaviour Models through Reusable, Multilevel, Coupled Model Transformations,” in *Proceedings of NWPT 2016*, P. Pettersson and W. Yi, Eds., 2016, pp. 49–51.

[25] “CPN tools,” http://cpntools.org/.

[26] T. Clark and J. Warmer, *Object Modeling With the OCL: The Rationale Behind the Object Constraint Language*. Springer, 2003, vol. 2263.

[27] C. Atkinson, R. Gerbig, and C. Tunjic, “Towards multi-level aware model transformations,” in *International Conference on Theory and Practice of Model Transformations*. Springer, 2012, pp. 208–223.

[28] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, “Generic model transformations: Write once, reuse everywhere,” in *ICMT Conference*, 2011, pp. 62–77.

[29] J. de Lara, E. Guerra, M. Chechik, and R. Salay, “Model transformation product lines,” in *Proceedings of the 21th MODELS conference*. ACM, 2018, pp. 67–77.

[30] E. Guerra, J. de Lara, M. Chechik, and R. Salay, “Analysing meta-model product lines,” in *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 2018, pp. 160–173.

[31] G. Perrouin, M. Amrani, M. Acher, B. Combemale, A. Legay, and P.-Y. Schobbens, “Featured model types: towards systematic reuse in modelling language engineering,” in *2016 IEEE/ACM 8th MiSE*. IEEE, 2016, pp. 1–7.

[32] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, “Atl: A model transformation tool,” *Science of computer programming*, vol. 72, no. 1-2, pp. 31–39, 2008.

[33] T. Degueule, B. Combemale, A. Blouin, O. Barais, and J.-M. Jézéquel, “Melange: A meta-language for modular and reusable development of dsls,” in *Proceedings of the 2015 SLE Conference*. ACM, 2015, pp. 25–36.

[34] H. Krahn, B. Rumpe, and S. Völkel, “Monticore: a framework for compositional development of domain specific languages,” *CoRR*, vol. abs/1409.2367, 2014.

[35] F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler, “On language-independent model modularisation,” *Trans. Aspect-Oriented Software Development*, vol. 6, pp. 39–82, 2009.

[36] F. Durán, A. Moreno-Delgado, F. Orejas, and S. Zschaler, “Amalgamation of domain specific languages with behaviour,” *Journal of Logical and Algebraic Methods in Programming*, vol. 86, no. 1, pp. 208–235, 2017.