An Approach to Flexible Multilevel Modelling

Fernando Macías[*,a], Adrian Rutle[a], Volker Stolz[a], Roberto Rodriguez-Echeverria[b], Uwe Wolter[c]

[a] Western Norway University of Applied Sciences, Norway
[b] University of Extremadura, Spain
[c] University of Bergen, Norway

Abstract. *Multilevel modelling approaches tackle issues related to lack of flexibility and mixed levels of abstraction by providing features like deep modelling and linguistic extension. However, the lack of a clear consensus on fundamental concepts of the paradigm has in turn led to lack of common focus in current multilevel modelling tools and their adoption. In this paper, we propose a formal framework, together with its corresponding tools, to tackle these challenges. The approach facilitates definition of flexible multilevel modelling hierarchies by allowing addition and deletion of intermediate abstraction levels in the hierarchies. Moreover, it facilitates separation of concerns by allowing integration of different multilevel modelling hierarchies as different aspects of the system to be modelled. In addition, our approach facilitates reusability of concepts and their behaviour by allowing definition of flexible transformation rules which are applicable to different hierarchies with a variable number of levels. As a proof of concept, a prototype tool and a domain-specific language for the definition of these rules is provided.*

Keywords. Multilevel modelling • Model transformation • Graph transformation

Communicated by U. Frank. Received 2016-12-12. Accepted after 2 revisions on 2017-11-27.

1 Introduction

Model-Driven Software Engineering (MDSE) is a means for tackling the increasing complexity of software development processes through abstractions (Brambilla et al. 2012). In MDSE, the main aspects of software are usually modelled using mainstream modelling approaches which conform to the OMG standards, such as ECLIPSE MODELLING FRAMEWORK (EMF) (Steinberg et al. 2008). These approaches provide high reliability, a modelling ecosystem and good tool coverage. However, they are usually restricted to a fixed number of modelling levels, consequently suffering from mixed levels of abstraction, the need to encode a synthetic typing relation among model elements in the same level (Kühne and Schreiber 2007), and the difficulty to maintain and understand such models due to convolution.

Multilevel modelling (MLM) addresses the issues of fixed-level metamodelling approaches by enabling an unlimited number of levels, deep hierarchies, potencies and linguistic metamodels and extensions. Employing a multilevel modelling stack has proven to be the best approach in many application scenarios (Juan de Lara et al. 2014), not to mention its resemblance to real world domains, which are not necessarily restricted to a certain number of abstraction levels. The idea of multiple levels of abstraction applied to MDSE started to gain momentum in the 2000s (Atkinson and Kühne 2001a,b; Atkinson and Kühne 2002,

* Corresponding author.
E-mail. fmac@hvl.no
The authors thank Francisco Durán and Malte Schmitz for their comments and insightful discussions before and during the process of writing this paper, as well as the reviewers for their accurate suggestions for improvement in an earlier draft of this paper.

2008; de Lara and Vangheluwe 2010; Rossini et al. 2014), however, using several abstraction layers was already a well-established technique for the specification of information systems. Some examples are Atkinson (1997), Bézivin and Lemesle (1997), Borgida et al. (1984), Mylopoulos et al. (1990, 1980) and Odell (1994, 1998).

Nevertheless, multilevel modelling also has several challenges which hamper its wide-range adoption, such as lack of a clear consensus on fundamental concepts of the paradigm, which has in turn led to lack of common focus in current multilevel tools. As response to these challenges, in Atkinson and Kühne (2008), de Lara and Guerra (2010a) and Rossini et al. (2014), the authors have proposed formalisations of multilevel modelling—sometimes also called *deep* metamodelling. In this paper, we revise some of these concepts. So, in order to define the scope of this paper, we introduce and briefly explain how this work relates to other MLM approaches in the following.

One of the most mature approaches for multilevel metamodelling is MELANEE (Atkinson and Gerbig 2016), which is based on the concept of *clabject* (Atkinson 1997). While this framework has some similarities and shares some of our goals, we take a different approach in some of the formal aspects. Furthermore, our tools aim at integrating seamlessly into EMF, instead of creating a new set of modelling tools from scratch. Similarly, we share some commonalities with the tool METADEPTH (de Lara and Guerra 2010a), but redefine some of the concepts in the interest of high flexibility in the specification of behavioural languages—models—and semantics—model transformations. This work is conceived as an evolution of the DIAGRAM PREDICATE FRAMEWORK (Rutle et al. 2012). In Sect. 6 we discuss these three works, as well as other related approaches.

In this paper we propose a revised conceptual framework that tackles some of the open challenges of the MLM community. Some examples of these challenges are the usage and conceptual meaning of linguistic metamodels, the nature of the elements that conform to the models, the possibility of having more than one type for each element, or how to specify model transformations (MT) in a MLM environment. The formalisation presented in this paper aims at covering the fundamental aspects of MLM, especially those regarding flexibility of the specification. Furthermore, we show some practical applications of such a framework for MLM and MLM model transformation specification. Our focus is reusability, hence the stress on flexibility in every aspect of the framework: design of flexible MLM hierarchies, reuse of multilevel model transformation and combination or integration of different independent hierarchies so that common structure and behaviour could be reused. We apply these concepts in the field of Domain Specific Modelling Languages (DSML) for behavioural modelling. The concept of DSML is that of a Domain Specific Language (DSL) (Fowler 2011; Kelly and Tolvanen 2008), specified by means of modelling techniques.

In the first sections of this paper, we provide mathematical definitions for central concepts in multilevel modelling. Our objectives are: (1) to formalise these concepts, giving a concise and precise meaning to them and checking their consistency prior to implementation; (2) to establish a common conceptual ground for researchers, tool developers and model designers, who come from different fields, to discuss and find solutions for the problems addressed by the paper; and (3) to build a bridge to the expertise of theoreticians and practitioners in other relevant areas, like graph transformations. Our definitions and formalisation are based on graphs, which are the basic structure shared by most kinds of models. This is also a paradigmatic choice: our mathematical formalisation will enable us to convey, in a well-structured way, our ideas, definitions, and results to more complex graph-based structures, such as attributed graphs or E-graphs (Ehrig et al. 2006, 2004).

We also present in this paper prototype implementations of the main two aspects of the framework, namely building modelling hierarchies and specifying model transformations in a multilevel setting. Hence, the contributions that we present

in this paper are: (1) a formal framework for flexible MLM, presented in Sects. 2 and 3, (2) formal multilevel model transformations used for definition of behavioural semantics, presented in Sect. 4, and (3) two tools that realise these formal framework and techniques: the MULTECORE tool to build modelling hierarchies and a textual editor for the specification of such MT, in Sect. 5.

2 Multilevel Metamodelling Hierarchies

In this paper, we choose to focus on behaviour since the definition of behavioural metalanguages (level 0), languages (level 1), specific models defined using these languages (level 2) and the state of such models through time (level 3) provides a scenario which inherently requires multiple levels of abstraction. This comes in addition to the fact that organising the metalanguages and the languages in a multilevel hierarchy (i. e., through modularisation and locating abstract concepts and their behaviour at higher levels and their refinement and customisation to fit specific domains at lower levels) would enhance reusability. Due to this, some definitions in this section might seem to be more generic than required. However, we want to leave the door open for multilevel model-driven engineering and multilevel model transformations in general. Similarly, the goal of the defined model transformations is to serve as execution semantics for the behavioural languages that the models define, as the examples illustrate.

The building blocks of our proposed framework are models. Our models are represented by means of graphs, since graphs are widely used to represent software models due to the fact that graphs are a very natural way of explaining complex situations on an intuitive level, e. g. for data and control flow diagrams, for entity relationship and UML diagrams, for visualization of software and hardware architectures, etc. (Ehrig et al. 2006). These graphs are organised in a hierarchical manner. In this section, we introduce the kind of graphs used to represent our models, as well as the way to organize them in hierarchies, by means of relations among graphs and the elements—nodes and arrows—that they contain. We use an excerpt of the case study presented later on in the paper as an illustrative, incremental example. This example defines concepts from the domain of process languages, applied to the definition of behaviour for simple, autonomous robots using the Lego EV3 and Arduino platforms (Banzi 2008; Monk 2011). Note that some of the names in the example are shortened for display purposes when compared with the full version presented in Sect. 3.1.

The terminology used in some MLM approaches may vary, depending on the authors (Gerbig et al. 2016). In order to avoid confusion, we will use the following terms, related to the fact that we internally represent models as directed multigraphs (see Sect. 2.1):

- For every representation of a concept in a model, we will use the word *node*, since they are represented internally as graph nodes. This concept is named *clabject* in METADEPTH and MELANEE, or also *entity* in the latter.
- For every relation between two nodes, represented as a graph arrow—sometimes also named *edge*—we will use the term *arrow*.
- We do not use a special representation for attributes and their data types, but represent them as nodes, in a similar fashion to Mylopoulos et al. (1990). See Sect. 3.3 for more details about attributes and data types.

2.1 Directed multigraphs

Our multilevel metamodelling approach is based on a flexible typing mechanism. Therefore, we will consider models abstractly as *graphs*, represented with a name, usually G. Specifically, we work with directed multigraphs. These graphs consist of nodes and arrows. A node represents a class, and an arrow represents a relation between two classes. Hence, an arrow always connects two nodes in the same graph, and any two nodes can be connected by an arbitrary number of arrows. Arrows with source and target in the same node (loops) are also allowed, and a node can likewise have any number of loops. We will use the word *element* to refer to both nodes and arrows, and assume that all

elements are named and identified by such name. For this reason, the names of any two nodes in the same graph must be different. For the arrows, we allow for equal names as long as either the source, the target or both are different, in order to be able to differentiate them. Hence, the arrows are considered, in such a way, as triples (s, a, t) with a an arrow name and s, t the names of its source and target nodes, respectively. For a given graph G, we denote by G^N its set of nodes, and by G^A its sets of arrows. Furthermore, a graph requires for its definition two mappings $sc^G : G^A \to G^N$ and $tg^G : G^A \to G^N$ that assign to each arrow its source and target node, respectively. These two morphisms must be total for the graph to be considered valid. That is, we can define a graph as a quadruple $G = (G^N, G^A, sc^G, tg^G)$. Figure 1 shows a small graph named `robot_1`. This graph contains three nodes, `I`, `T` and `GF`, represented as yellow squares. The arrows connecting these nodes are labelled with their names, `in` and `out`.

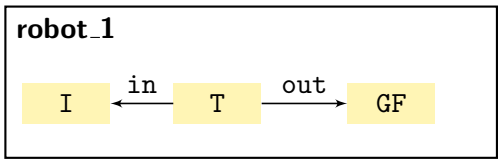


Figure 1: Directed multigraph with named elements

The relations between graphs, like typing and matching—explained in the next sections —, are defined by means of *graph homomorphisms*. A homomorphism ϕ from graph G to graph H is given by two maps $\phi^N : G^N \to H^N$ and $\phi^A : G^A \to H^A$ that are compatible with the involved source and target maps, respectively.

Due to the use of graphs as the underlying representation of models, both terms 'graph' and 'model' could be used interchangeably in this paper. We will however differentiate by using the former for the definitions in this section, and the latter when discussing examples in the next ones.

2.2 Tree shaped hierarchies, abstraction levels and typing chains

We assume that our graphs (models) are organized in a tree-shaped hierarchy with a single root graph (in the sequel, it becomes clear that we will indeed get a tree-shaped hierarchy since typing is unique within each hierarchy). Implicit in that assumption is the fact that each graph, except the one at the root, has exactly one parent graph in the hierarchy. Also, we allow for arbitrary finite branching in the tree, so that each graph can have none or arbitrary finite many sibling graphs.

The hierarchy has $l+1$ *abstraction levels* where l is the maximal length of paths in the hierarchy tree. Each level in the hierarchy represents a different degree of abstraction. Levels are indexed with increasing integers starting from the uppermost one, with index 0. Each graph in the hierarchy is placed at some level i, where i is the length of the path from the root to the graph. We will use the notation G_i to indicate that a graph is placed at level i. Level 0 contains, in such a way, just the root G_0 of the tree. For any graph G_i we call the unique path $TC(G_i) = [G_i, G_{i-1}, \ldots, G_1, G_0]$ from this graph to the root graph of the hierarchy the *typing chain* of G_i.

Possible candidates for the root graph G_0 are Ecore (Steinberg et al. 2008) and the

graph. For implementation reasons, we use Ecore as root graph in all hierarchies since Ecore is based on the concept of graph which makes it powerful enough to represent the structure of software models. Although it might be more appropriate, conceptually, to use Ecore as a linguistic metamodel (Atkinson and Kühne 2001b), we leave Ecore on top of our hierarchy since (i) Ecore does not provide any concepts for defining levels and typing between levels and (ii) our implementation strives to minimise the threshold for migration from fixed-level EMF-based modelling to MLM. While the modelling hierarchies presented in this paper have a mixture of linguistic and ontological nature, it is out of the scope of this paper to discuss the relations between ontological and linguistic hierarchies and their purposes. Nonetheless, these hierarchies are not necessarily ontological since they may also exist as prescriptive models for software implementation (Aßmann et al. 2006). It is also important to note that, in the actual

status of implementation, level numbers are replaced by references from a model to its immediate metamodels, since the numbers are only needed for formal constructions. In the hypothetical case where these numbers were introduced and needed to be updated when new levels are added on top of them, traversing the hierarchy and updating them would be automatic and almost trivial.

In Fig. 2, we show several graphs that constitute a tree-shaped hierarchy, included the one already presented. We locate the graph `robot_1`, representing a particular configuration for a robot, in level 3, together with another graph `robot_2`, defining a different configuration for another robot. These two graphs, although conceptually similar, are independent from each other in the sense that they neither belong to the same branch of the tree nor share their parent graph—that is, their metamodel. They serve the role of parent graphs, respectively, for `robot_1_run_1` and `robot_2_run_1`, located at level 4. Since we in this paper focus on behavioural modelling languages, the models we define evolve through time, representing the execution of the modelled system. Thus, the purpose of `robot_1_run_1` is to store, at any particular point in time, the state of the execution of the specific process defined in `robot_1`; likewise for `robot_2_run_1` and `robot_2`.

Back to our hierarchy, the parent graphs of `robot_1` and `robot_2` are located at level 2, where most of the types of their elements are defined. These two parent graphs, `legolang` and `arduinolang`, define the type of elements that we can use to define a specific process for a robot designed in the Lego EV3 or Arduino platforms, respectively. Both models share the common parent graph `robolang`, located at level 1. This language contains basic concepts for process modelling, independently from the robot hardware or platform. Finally, as we previously mentioned, we locate Ecore, the topmost graph and root of the tree, at level 0. This graph is the parent graph of `robolang`.

It could be argued that using specialisation and generalization (Borgida et al. 1984; Kühne 2009; Mylopoulos et al. 1980) could reduce this whole hierarchy into two levels. While it is outside this paper's scope to illustrate the advantages of MLM as an alternative or to take a stand on the classification vs generalization question, we enumerate below some arguments for why we have chosen MLM for this example.

- Separation of concerns: robots which do not have certain capabilities—say, flying—would not need a language with `FlyUp` and `FlyDown` tasks, hence specialisation of `Task` in a single level would pollute the metamodel and weaken the domain-specificness of the languages.
- Reusability: concepts and behaviour defined at a higher level can be reused or overridden at lower levels in the hierarchy.
- Extendibility: extending a metamodel with other elements in the future would give some challenges wrt. modification of editors and other artefacts which are created from that metamodel, while adding a new metalevel below (or above) makes the extension independent on other models and does not affect past or future updates of other artefacts.
- Modularization: each metamodel in the hierarchy can be used as a module for itself without paying much considerations to neighbouring branches.
- Specification of behaviour: The approach that we present here would be infeasible in 2-level modelling. If the whole language for robots is collapsed into one level, the instances and their states would also have to exist in one single model. Defining behaviour by means of model transformations in such a scenario would require more complex rules to keep consistent the defined behaviour and current state by other relations than typing, polluting the model, and leading to a bigger set of rules, as illustrated in Sect. 4.

In Fig. 2, we use red horizontal lines to indicate the separation between two levels, and blue dashed arrows indicating the sequences of graphs that constitute typing chains and provide the required tree shape. Note that the contents of both graphs

`arduinolang`, `robot_2` and `robot_2_run_1` are not displayed since they are not relevant for this example, but serve to illustrate the tree shape. The diagram with the full left branch of the hierarchy is displayed in Fig. 7, where we also explain in more detail the concepts defined in the graphs.

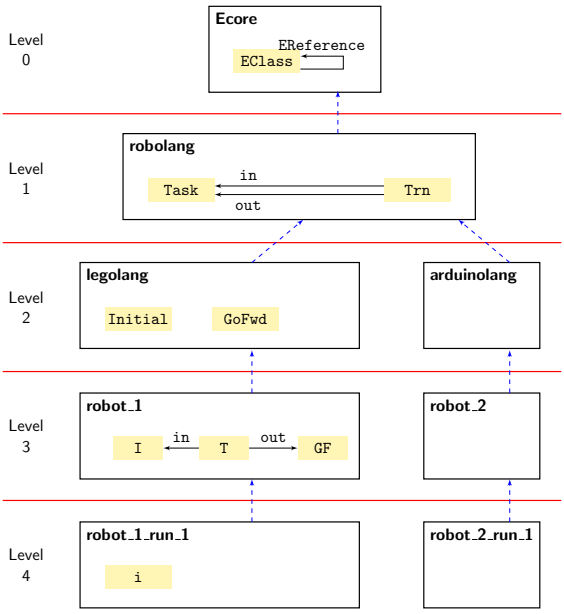


Figure 2: Tree-shaped hierarchy of graphs with levels and typing chains

2.3 Individual typing

Any element `e` in any graph G_i has a unique type denoted by $ty(\mathtt{e})$. In that case, we can say that `e` is *typed by* $ty(\mathtt{e})$ or, equivalently, that `e` is an *instance* of $ty(\mathtt{e})$. The $ty(\mathtt{e})$ has the semantic meaning of the 'class' of `e`, e. g. $ty(\mathtt{Task}) = \mathtt{EClass}$. Although this relation, i. e. classification, is conceptually different from specialisation (Kühne 2009), we do not put any restrictions on how a model designer would organise the concepts in her hierarchy, and leave room for the definition of good and bad practises in a similar fashion to object-oriented patterns and anti-patterns. Hence, some modelling engineer may consider that $ty(\mathtt{Initial}) = \mathtt{Task}$ should be replaced by specialisation, although we chose the former for flexibility and illustration purposes.

To achieve the necessary flexibility, we allow typing to jump over levels. That is, for any `e` in a graph G_i, with $i \geq 1$, its *individual type* $ty(\mathtt{e})$ is found in a graph $TG(\mathtt{e})$, which is one of the graphs $[G_{i-1}, \ldots, G_1, G_0]$ in the corresponding typing chain. Note that the types of different elements in G_i may be located in different graphs. By $df(\mathtt{e})$ we denote the difference between i and the level where $TG(\mathtt{e})$ is located. In most cases, this difference is 1, meaning that the type of `e` is located at the level directly above. In short, we can say that, for any element `e` in a given graph G_i, with $i \geq 1$, its type $ty(\mathtt{e})$ is an element in the graph $G_{i-df(\mathtt{e})}$, where $1 \leq df(\mathtt{e}) \leq i$.

Figure 3 displays our example hierarchy, including the type of each element. To avoid polluting the diagram with too many arrows, we use alternative representations for them. For every node, its type is identified by name and depicted in a blue ellipse attached to the node. For example, the type of `I` is `Initial`, which could also be represented as an arrow between these two nodes. For the arrows, the type is represented as another label with the name of the type. This label is distinguished from the one with the element's own name by using italics font. For example, the type of the `out` arrow in `robolang` is `EReference`, located in the `Ecore` graph.

To ensure that every element has a type, we assume that the root graph G_0 has a *self-defining* collection of individual typing assignments. In other words, we have that if $\mathtt{e} \in G_0$ then $ty(\mathtt{e}) \in G_0$. We use the more relaxed term *self-defining* instead of *reflexive* since the former just requires that all the elements in the graph are typed by elements in the same graph, whereas the latter means that every element is strictly typed by itself.

From a more general point of view, we obtain for any `e` in G_i a sequence

$$\begin{array}{ccccccc} \mathtt{e} & \mapsto ty(\mathtt{e}) & \mapsto ty^2(\mathtt{e}) & \mapsto \cdots & \mapsto ty^{s_\mathtt{e}}(\mathtt{e}) \\ \cap & \cap & \cap & & \cap \\ G_i & G_{i-df(\mathtt{e})} & G_{i-df^2(\mathtt{e})} & & G_{i-df^{s_\mathtt{e}}(\mathtt{e})} = G_0 \end{array}$$

of typing assignments of length $1 \leq s_\mathtt{e} \leq i$ with $(i - df^{s_\mathtt{e}}(\mathtt{e})) = 0$. The number $s_\mathtt{e}$ of steps depends

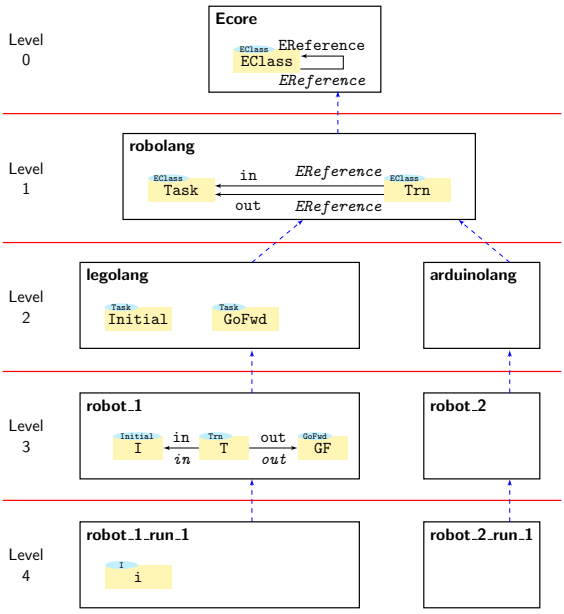


Figure 3: Graph hierarchy with typed elements

individually on the item e. For convenience, we use the following abbreviations:

$$ty^2(\mathtt{e}) = ty(ty(\mathtt{e}))$$

$$ty^3(\mathtt{e}) = ty(ty(ty(\mathtt{e})))$$

$$\ldots$$

$$df^2(\mathtt{e}) = df(\mathtt{e}) + df(ty(\mathtt{e}))$$

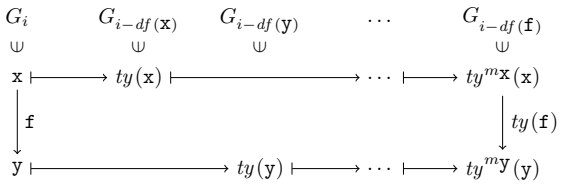
$$df^3(\mathtt{e}) = df^2(\mathtt{e}) + df(ty^2(\mathtt{e}))$$

$$\ldots$$

Let us consider an arbitrary arrow $\mathtt{x} \xrightarrow{\mathtt{f}} \mathtt{y}$, together with its source and target nodes, in a graph G_i (note that we will just use f to refer to that particular arrow, to simplify the notation). It may happen that the types of the three elements are located in three different graphs; e. g. the edge in in the graph at level 3 in Fig. 3 has its type at level 1, while the type of its source is also at level 1, the type of its target is at level 2. The typing of arrows should, however, be compatible with the typing of sources and targets. A natural requirement would be that the source and the target of $ty(\mathtt{f}) \in G_{i-df(\mathtt{f})}$ are provided by the type of x and the type of y, respectively; i. e., referring to the above mentioned example, the type of the type of the target of in is also located at level 1 and the edge is indeed allowed between these two types. Specifically, we require that the following *non-dangling typing* condition is satisfied: There exist $1 \leq m_{\mathtt{x}} \leq s_{\mathtt{x}}$ and $1 \leq m_{\mathtt{y}} \leq s_{\mathtt{y}}$ such that

- $df^{m_{\mathtt{x}}}(\mathtt{x}) = df^{m_{\mathtt{y}}}(\mathtt{y}) = df(\mathtt{f})$ and
- $ty^{m_{\mathtt{x}}}(\mathtt{x})$ is the source of $ty(\mathtt{f})$ and
- $ty^{m_{\mathtt{y}}}(\mathtt{y})$ is the target of $ty(\mathtt{f})$.

Hence, the typing chains of the arrow and its source and target nodes must look as follows:



2.4 Typing morphisms and domains of definition

Our notions of flexible multilevel typing can be described for graphs—hence, in a more abstract and compact way—by means of a family of typing morphisms. That is, we can define typing relations between any two graphs, in a typing chain, as an abstraction of the individual typing that we define for their elements in Sect. 2.3. The vocabulary defined for individual typing can be reused here, so that a graph can be an *instance* of another graph, or be `typed by` it. These relations among graphs are defined by means of graph homomorphisms. Since the individual typing maps are allowed to jump over levels, two different elements in the same graph may have their type located in different graphs along the typing chain. Hence, the typing morphisms established between graphs become partial graph homomorphisms, cf. Rossini et al. (2014).

Our characterization of individual typing ensures that, for any levels i, j such that $0 \leq i < j \leq l$, there is a *partial typing morphism* $\tau_{j,i} : G_j \multimap G_i$ given by a subgraph $D(\tau_{j,i}) \sqsubseteq G_j$, called the *domain of definition* of $\tau_{j,i}$, and a *total typing homomorphism* $\tau_{j,i} : D(\tau_{j,i}) \to G_i$. In

abuse of notation, we use the same name for both morphisms since they represent the same typing information. Using the same syntax as the examples, Fig. 4 depicts these concepts in a generic manner.

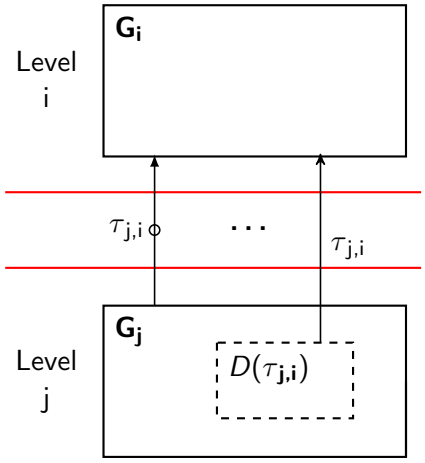


Figure 4: Typing morphisms and domain of definition

Given that a graph is given by its set of nodes and its set of arrows, we can define the concept of domain component-wise. For any $0 \leq i < j \leq l$ we can define the set $D(\tau^N_{j,i}) \subseteq G^N_j$ of all nodes in G_j that are recursively typed by nodes in G_i. That is, $\mathsf{e} \in D(\tau^N_{j,i})$ iff there exists $1 \leq m \leq s_{\mathsf{e}}$ such that $j - i = df^m(\mathsf{e})$, and thus $ty^m(\mathsf{e}) \in G^N_i$. We set $\tau^N_{j,i}(\mathsf{e}) := ty^m(\mathsf{e})$ for all nodes in $D(\tau^N_{j,i})$ and obtain, in such a way, a morphism $\tau^N_{j,i} : D(\tau^N_{j,i}) \to G^N_i$. This total morphism defines a partial morphism $\tau^N_{j,i} : G^N_j \multimap G^N_i$ with the domain of definition $D(\tau^N_{j,i})$.

We can follow an analogous process for the set of arrows G^A to obtain a partial morphism $\tau^A_{j,i} : G^A_j \multimap G^A_i$ with domain of definition $D(\tau^A_{j,i})$.

The non-dangling typing condition is now equivalent to the requirement that the pair $(D(\tau^N_{j,i}), D(\tau^A_{j,i}))$ constitutes a subgraph $D(\tau_{j,i})$ of G_j. Consequently, the pair of morphisms $(\tau^N_{j,i}, \tau^A_{j,i})$ provides a total graph homomorphism $\tau_{j,i} : D(\tau_{j,i}) \to G_i$ and thus a partial typing morphism $\tau_{j,i} : G_j \multimap G_i$.

All the typing morphisms $\tau_{j,0} : G_j \to G_0$ are total, since every element has a type. The uniqueness of typing is reflected on the abstraction level of type morphisms by the *uniqueness condition*: For all $0 \leq i < j < k$, we have that $\tau_{k,j}; \tau_{j,i} \preceq \tau_{k,i}$. The symbol $\preceq$ means whenever an element in G_k is recursively typed by an element in G_j such that this element in G_j is, in turn, again recursively typed by an element in G_i then the element in G_k is also recursively typed by an element in G_i and both ways provide the same type in G_i.

The other way around, we can reconstruct individual typing from a family of partial typing morphisms between graphs that satisfy the totality and the uniqueness condition. For any item e in a graph G_k there exists a maximal, least abstract level $0 \leq i_{\mathsf{e}} < k$ such that e is in $D(\tau_{k,i_{\mathsf{e}}})$ but not in $D(\tau_{k,j})$ for all $i_{\mathsf{e}} < j < k$, since $\tau_{j,0}$ is total and k a finite number. Hence, the individual type of e is given by $ty(\mathsf{e}) := \tau_{k,i_{\mathsf{e}}}(\mathsf{e})$ and $df(\mathsf{e}) := k - i_{\mathsf{e}}$.

In Fig. 5, we represent the typing morphisms between all the graphs in the hierarchy. Note that the morphism from `robolang` to `Ecore` is total, since all the types used in the former can only be defined in the latter, and all elements must have a type. Moreover, it is possible to define a total morphism from any graph in the hierarchy to the root graph via composition of partial typing morphisms with respect to the individual typing morphisms that they represent.

In summary, we define multilevel typing, first, by individual typing of graph elements, and then by abstracting it away by means of partial typing morphisms. This more abstract and concise definition of multilevel typing can be conveyed straightforwardly to other structures, and we have outlined a proof showing the equivalence of both views for graphs. In this section, we have used the definitions to identify precisely which conditions the typing relations should satisfy, e. g. the uniqueness condition and the non-dangling condition.

2.5 Potency

In this subsection, we introduce our modification and formalisation of the concept of *potency of*

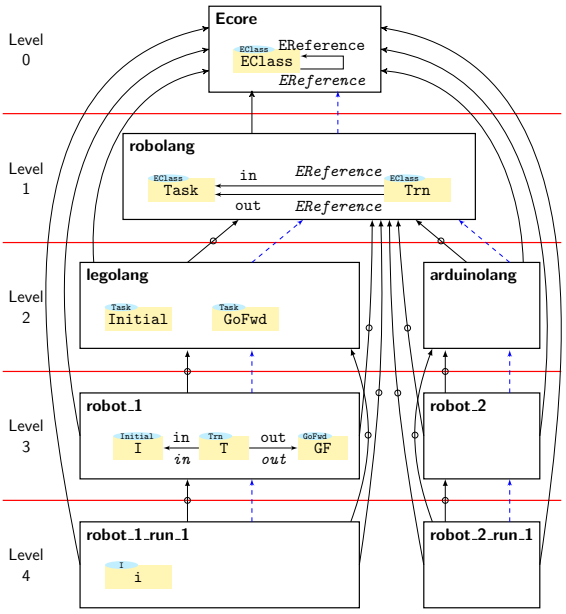


Figure 5: Graph hierarchy with typing morphisms

an element originally defined in Atkinson and Kühne 2002. Potency is used on elements as a means of restricting the length of the jumps of their typing morphisms across several levels. The reason for this is that the formalisation presented so far is aimed at classifying graphs in tree-shaped hierarchies as a means to get a clear structure of the concepts defined, but the hierarchy is built based on the individual typing relations, which do not have any restrictions regarding levels. Due to this fact, levels become less useful in practice if the individual typing relations are unbounded, hence the necessity of potencies to restrict the jumps of typing morphisms across levels.

Other authors represent the potency of an element with just a number, whereas we employ an interval that allows for a higher degree of expressiveness, using the notation `min--max`. These values may appear after the declaration of an element, using `@` as a separator. In the cases where `min` = `max`, the notation can be simplified to show just one number. Such is the case with the default value of potency: `@1-1` ≡ `@1`. Furthermore, some existing realizations of potency have different effects on classes than on attributes. However, we do not require to differentiate between them since attributes and their data types are also represented as nodes and arrows (see Sect. 3.3).

The formal definition of potency as a range is as follows. For any element `x` in G_i we require $min \leq df(\texttt{x}) \leq max$ where $ty(\texttt{x})$`@min--max` is the declared potency of the type $ty(\texttt{x})$ in $G_{i-df(\texttt{x})}$. This condition can be reformulated using partial typing morphisms. For any element `y` in G_j with a potency declaration `y@min--max` we require that $\tau_{j,i}^{-1}(\texttt{y})$ is empty for all j with $j - i <$ `min` or $j - i >$ `max`. In all other cases, that is, `min` $\leq j - i \leq$ `max`, there is no strict requirement. $\tau_{j,i}^{-1}(\texttt{y})$ may be empty or not.

For the typing relations used in our example to be correct, and given that we assume the default potency `@1--1` unless otherwise specified, we require the node `Trn` and the arrows `in` and `out` in `robolang` to have increased potency (see Fig. 6). That way, it becomes possible to use them as types, respectively, for `T`, `in` and `out`; located *two levels below*, in the `robot_1` graph. Besides, and to ease the interpretation of a diagram, the type of an element which is specified in a level different from the one immediately above reuses the `@` notation to indicate it. For nodes, the annotation is located in the blue ellipse. For arrows, the `ty(e)@`df`(e)` annotation is displayed in the declaration of the type (in italics). In both cases, we also define a non-graphical representation of the element and its type in the form `e:ty(e)@`df`(e)`. For the default value `1`, used in all the other cases, the notation `e:ty(e)` is used as a shorthand for `e:ty(e)@1`. Figure 6 shows the final version of the example hierarchy in which potencies have been added to the elements aforementioned.

Note that the potencies of the Ecore elements `EClass` and `EReference` are not the default `1--1`, defined for 'user-accesible' levels but, `0--*`. The minimum potency of `0` is required to allow for self-definition and the maximum value of `*`—unlimited—is required to be able to create direct instances of Ecore elements at any level below, without forcing the creation of an intermediate type.

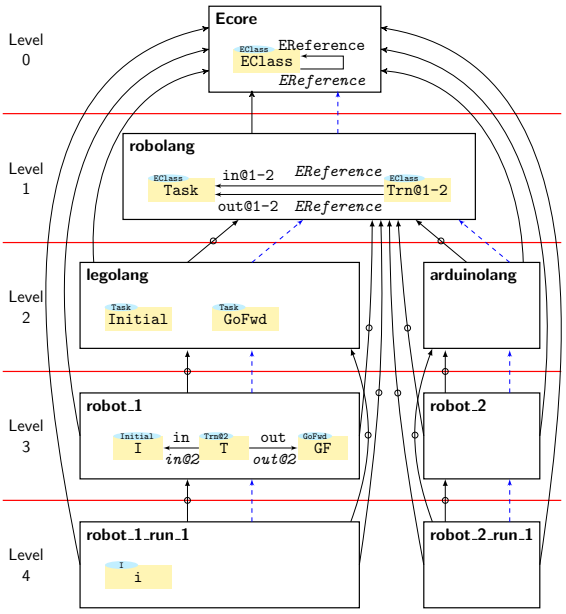


Figure 6: Full graph hierarchy with potency

2.6 Inheritance and multiplicity

These two techniques allow to further control the processes of specification and further instantiation of the elements defined in a model.

The inheritance (i. e. specialisation) relation is a special type of arrow among any two nodes within the same level, which imposes on the child node the same typing and potency as the parent node. Moreover, the inheritance relation gives the child node access to the incoming and outgoing arrows of the parent node, while still allowing the child node to define additional attributes or arrows. However, in order to avoid conflicts, the child node cannot redefine an arrow of its parent node by reusing its name. Using the concepts defined in the previous subsections, we can reformulate this requirement as follows: for any two nodes $\mathtt{x}, \mathtt{y} \in G_j^N$ and $0 \leq i < j$, if $\mathtt{x}$ inherits from $\mathtt{y}$, then it is required that $\tau_{j,i}^N(\mathtt{x}) = \tau_{j,i}^N(\mathtt{y})$. Furthermore, cyclic inheritance and inheritance between arrows are forbidden.

In the case of multiplicity (i. e. cardinality) of arrows, we consider them as an additional annotation that can be added on top of the arrow to further control the instantiation process. That is, the formalisation presented above does not constrain the amount of direct instances of an arrow that can be created. For this purpose, we introduce the concept of multiplicity of arrows. So, the same way as potency allows the modeller to control *where* instantiation can happen, multiplicities serve the purpose of specifying *how many times* a direct instance of an arrow can, or should, be created. Multiplicities are expressed in this work in the traditional notation of `min..max`, where the `min` (`max`) value restricts the minimum (maximum) number of direct instances of the arrow in the context of its source node. The default value for multiplicity is `0..n`, where `n` represents *unbounded*.

Both techniques are straight-forward and directly enforceable in the tool implementation.

3 A Three-Dimensional Framework

All the concepts defined in Sect. 2 are used to specify a single hierarchy that contains a family of related models with different levels of abstraction but a common domain—like Robolang, used in the examples so far. Nevertheless, complex scenarios from behavioural modelling may comprise more than one domain. For example, it could be useful to introduce verification aspects into the language, so that we can specify correctness properties, or enhance it with logging capabilities that gather information about the behaviour of a particular instance. In addition, data types—such as integers, strings or boolean values—are elements that may appear in any model and should be reusable in any hierarchy we create. These three 'aspects' of modelling—the base language, its possible additional aspects and the data types—can be represented as different modelling hierarchies, in a consistent manner, using the definitions from Sect. 2.

In this section, we introduce three dimensions where hierarchies can be located, depending on their nature and purpose, as well as the relations that can be defined between elements in different dimensions. First, we consider the hierarchies that define behavioural models and instances—since

that is our focus in this paper—to be the 'main' ones. We define these hierarchies to be located in the *application dimension* and, in consequence, we name them *application hierarchies*. These are presented in more detail in Sect. 3.1. Second, the hierarchies representing additional aspects of an application hierarchy are located in a secondary dimension. We call this dimension the *supplementary dimension*, and therefore we name the corresponding hierarchies as *supplementary hierarchies*. Section 3.2 delves into this concept and introduces an example of supplementary hierarchies. And third, we represent data types, commonly used for attributes in other modelling approaches, as yet another modelling hierarchy. This last hierarchy is fixed and consists of four levels, which are presented in Sect. 3.3. The *data type hierarchy* is the only one located in the *data type dimension*.

The relations between elements in different dimensions are expressed by means of *multiple typing*. This concept is defined as an extension of the individual typing presented in Sect. 2.3. If there are one or more supplementary hierarchies for a given application hierarchy, any element `e` defined in the application hierarchy may have several types, combined as follows:

- Exactly one typing map to another element in the same application hierarchy. In the uncommon case that this type is not specified, it will be assigned a default one—in our implementation using Ecore, the type would be `EClass` or `EReference` for nodes and arrows, respectively. That is, the requirement that every element has at least one type must hold.
- 0..n typing maps to supplementary hierarchies. Each hierarchy represents a different *aspect*, so the element `e` can have one supplementary type in each supplementary hierarchy related to the application hierarchy where `e` is located. Sect. 3.2 shows an example of such case and explains how to use supplementary typing, comparing our definition of multiple typing with the concepts of linguistic metamodels and extensions.
- 0..1 typing maps to the data type hierarchy. This is the case when declaring attributes. In Sect. 3.3 we illustrate how to apply single or double typing to represent attributes and their data types.

It is worth pointing out the fact that the typing map $\mathtt{e} \rightarrow ty(\mathtt{e})$, with `e` and $ty(\mathtt{e})$ belonging to the same application *hierarchy*, is the only one allowed in the application *dimension*. That is, the type of `e`—or any of its types—cannot be in a different application hierarchy than `e`. This means that any two application hierarchies are totally unrelated to each other, in the sense that it is not possible to create typing maps between their elements. In other words, there cannot be two application hierarchies in the same system.

Each one of the typing maps of `e` has its own typing chain associated. Hence, the concept of typing chains is also extended to allow the graphs that form it to belong to different hierarchies in different dimensions. For the sake of simplicity, the examples in this paper just illustrate cases with *double typing*, but there are no conceptual or technical obstacles, that we can foresee, to adding more than two types to an element in an application hierarchy.

3.1 Application dimension

The hierarchies used to define behavioural languages are located in this dimension. One of the defining characteristics of application hierarchies is their independence from the supplementary hierarchies. That is, while some scenarios cannot be modelled without using a combination of both kinds of hierarchies, the basic concepts of an application hierarchy should be independent of any supplementary one. For example, the temporal properties introduced in Sect. 3.2 require to double type application elements. However, those application elements can constitute a valid application model without the double type.

Besides, an element with multiple typing where one of the types belongs to an application hierarchy must always be inside an application hierarchy. That means that the supplementary and data type

dimensions always act as secondary or complementary aspects, and never as the main language of a specification. In a more general way, we can reformulate this as the requirement that any graph G, containing an element `e` ($e \in G^N$ or $e \in G^A$) with double type, will necessarily be part of an application hierarchy.

Furthermore, as pointed out in the introduction, we focus on the definition of behavioural models and their execution semantics specified as model transformations. The fact that the models in an application dimension are independent from those in the supplementary dimension is reflected also in their execution semantics. That is, the model transformations that define such semantics for application hierarchies should be independent from the supplementary models and model transformations.

Finally, also related to the previous point, in case of combining the behavioural semantics of different hierarchies (see Sect. 4), the application semantics are the last ones to be executed, since they are not aware of the influences of the other dimensions and will disregard any other semantics if executed in the first place. We discuss this issue in more detail in Sect. 7.

In Fig. 7, we display a hierarchy of models used for the definition of behaviour on simple robots, in order to demonstrate the properties and purpose of application hierarchies. This figure is the full version of the left branch of the hierarchy used as an example in Sect. 2. As before, we do not show Ecore on top of the hierarchy since it is not relevant for the explanation. The figure has been generated from the real implementation of the Robolang hierarchy created with the tool MultEcore, that we show in Sect. 5.1. For this reason, there is a slight change to the syntax used so far: every node has an additional decoration that displays its potency, instead of appending it to the name of the node. Also, the default value for the multiplicity of an arrow is `0..n`, which is not displayed for the sake of clarity.

In Fig. 7(a), we define a DSML for process modelling in the domain of robots, hence the name `robolang`. The main concept in this language is `Task`, which represents an action that a robot can perform. `Transitions` are used to connect any number of `in` tasks to any number of `out` ones. These transitions are triggered by `Inputs`, related to transitions by the `inputs` relation. Note that there may still exist instances of `Transition` without an associated instance of `Input`, in which case it gets fired instantaneously. For example, the ones leaving from initial tasks.

The concepts defined in the `robolang` language are specialised for simple Lego EV3 robots in `legolang`. This kind of robots are capable of moving in a flat surface, and detect both physical obstacles in their way and the borders of that surface. The `legolang` language, depicted in Fig. 7(b), defines specific tasks for the movement possibilities of such robots: `GoForward`, `GoBack`, `TurnLeft` and `TurnRight`. It also contains `Initial`, which represents the starting point in the process of performing those tasks. Besides, the language defines the required elements for the detection of a `Border` in the surface, an `Obstacle` in front of the robot, or the expiration of the time assigned to a task (`Timeout`). One of the key points in this model is the lack of arrows. The fact that tasks are related to each other (by transitions) is already specified in the model above, and that information is enough to specify the behavioural semantics of the language, as explained in Sect. 4.1.

In Fig. 7(c), we define a specific behaviour for a robot, named `robot_1`. The robot starts by executing its initial task `I`, which fires automatically the transition `T1` to `GF`, indicating the robot to go forward. Then, the robot may take two different courses of action. In the first one, it will fire `T2` and start going back (`GB1`) if an obstacle (`O`) is detected. After it goes back for a while, the timeout (`TO`) fires the transition (`T4`) and it starts turning left (`TL`). After another timeout, `T6` is fired, and the robot resumes to going forward again until another input is detected. The second course of action is similar to the previous one, but it starts by detecting a border (`B`), which fires `T3` and commands the robot to go back (`GB2`). After timeout, it will fire `T5` and turn right (`TR`) until the next timeout,

which will fire `T7` and return to `GF`, completing the loop. Note that all the arrows are typed by `in`, `out` and `input`, defined in `robolang`. As explained in Sect. 2, these arrows can be defined without requiring an intermediate redefinition in `legolang`, thanks to the use of potency. Thus, the separation in levels of abstraction is satisfied—it is not desirable that concepts from the level of abstraction of `robolang` have to be unnecessarily repeated in `legolang`.

The bottommost level on this hierarchy, depicted in Fig. 7(d), represents the state of a particular execution of the `robot_1` model, hence the name `robot_1_run_1`. The represented state is actually the initial state, where the robot begins its tasks by running an instance of the initial task of the model (`i:I`). This is the level that will be modified during the execution of the behavioural semantics defined as model transformations (see Sect. 4.1).

3.2 Supplementary dimension

The purpose of hierarchies defined in this subsection is defining additional aspects that can be introduced in behavioural languages.

Compared with other MLM approaches, like the ones mentioned in Sect. 1, the purpose of supplementary hierarchies can be understood as an evolution of the concept of *linguistic metamodels*, including 'linguistic typing' and 'linguistic extension', as follows. First, the inspiration for separate dimensions for different hierarchies of languages, and having one of them acting as the 'main' one, while the other adds some new concepts, stems from the idea of linguistic metamodels (Atkinson and Kühne 2001b; Rossini et al. 2014). In these models, some concepts are defined so that they can be used in the main ('ontological') hierarchy. These *linguistic types* must be applied on ontological elements, providing elements with double ontological-linguistic typing. This means that the ontological hierarchy is dependent on the linguistic metamodel in the sense that every ontological element requires a linguistic type, which is a consequence of employing the *clabject* paradigm. Alternatively, elements defined in a linguistic metamodel can be used as the only type of elements in the ontological hierarchy, hence creating elements with only one linguistic type and no ontological one. This technique is known as *linguistic extension*. Both concepts can be emulated in our conceptual framework by means of using an additional hierarchy, which serves the role of linguistic metamodel. In addition, our framework does not constrain other possibilities like an element having just ontological type—type on its own hierarchy—or more than two types—one additional type per supplementary hierarchy, while keeping the constraint of exactly one type per hierarchy. This also allows for addition of initially unforeseen types, as other modelling approaches do. See, for example, J. de Lara et al. (2015).

We illustrate one possible use of supplementary hierarchies in the following. In that example, we define a property specification language in order to describe temporal correctness properties for behavioural languages. As stated in Sect. 3.1, Ecore is the topmost level of the hierarchy from a formal point of view, although not user-accessible or modifiable.

The LTL (Linear-time Temporal Logic) specification language defines a propositional logic (Manna and Pnueli 1995). This implies that one of the elements in the language are atomic propositions which can be evaluated at any particular point in time to *true* or *false*. Since we apply this language to behavioural models, the result of the evaluation is based on the state of the model, that is, the existence of specific elements on the running instance of the model. The LTL language contains the usual boolean connectives and additional operators, which define the meaning of a formula over a trace of observations for the propositions.

In Fig. 8, we represent the model that defines all the LTL concepts, which will allow us to specify LTL properties. This kind of language has been previously used in model-based approaches applied to the field of runtime verification for the specification of embedded systems (Macías et al. 2016). In that work, the key contribution is that correctness properties become part of the

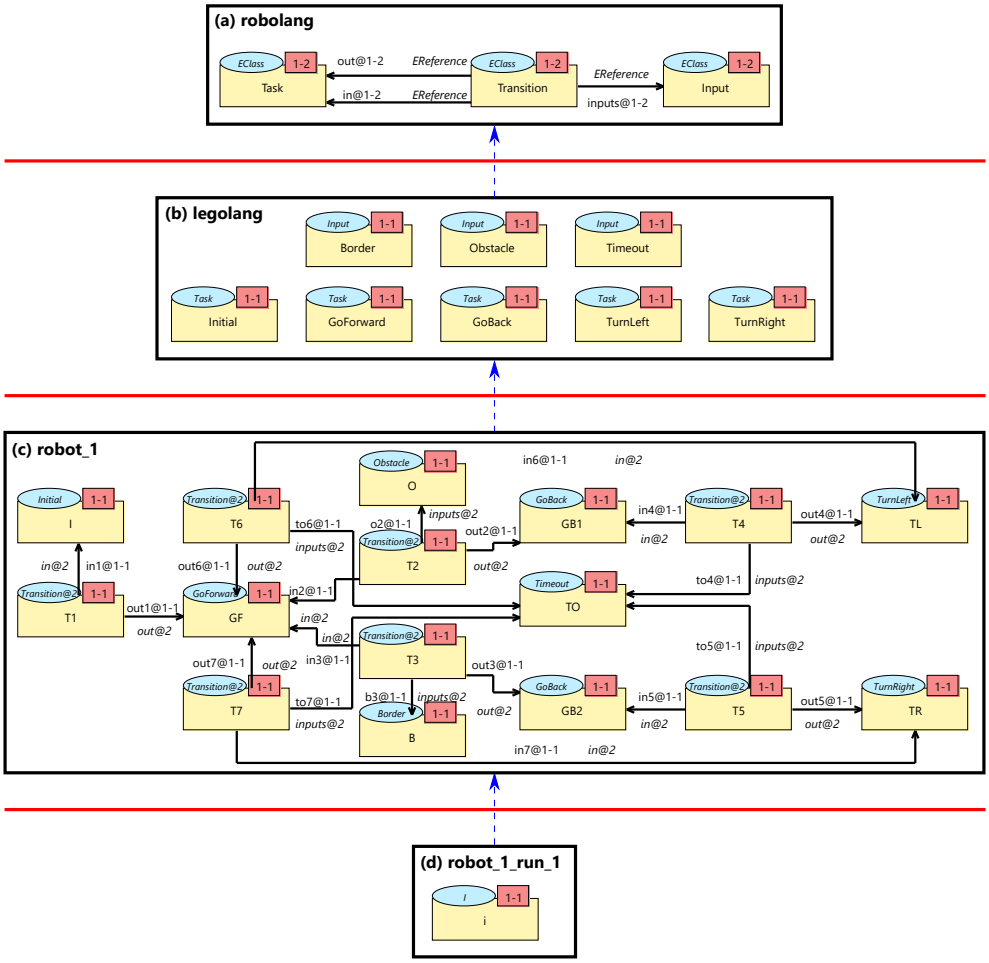


Figure 7: Full hierarchy for Robolang case study

modelling hierarchies to which they refer. On the one hand, the authors point out the advantage that the properties evolve together with the actual model they refer to, and on the other hand that the model can be used for simulation, code generation and evaluation of such properties. We used the same abstract syntax in Fig. 8 as in the rest of the examples, to show that our framework is capable of also representing the grammar and semantics of a second-order logic. In practice, it would be more convenient to specify a concrete—probably textual—syntax, synchronised with these abstract concepts.

The reasons to depict the single `ltl` model as a means to present this hierarchy are twofold. First, we omit the Ecore metamodel on top, as we do in all the other hierarchies presented in this section. And second, the properties are specified as new models typed by the elements in `ltl` but, as explained below, some of its elements will have a double typing relation (one application type and one supplementary type). In this case—of elements having double typing—, and as explained at the beginning of this section, the model will be located in the application hierarchy whose types are being used and, as a consequence, the models representing specific LTL properties will not be part of the LTL supplementary hierarchy.

In the model, all elements are instances of Ecore types, but this time we display that type in a green ellipse, instead of blue, to clarify that they belong to a supplementary hierarchy. The main

element is `Formula`, a node which can represent the whole LTL property, as well as any of its subexpressions, connected by operators. These operators can be `Unary` or `Binary`, in which case they are connected to one or two subformulas, respectively. These connections are represented with the relation `formula` in the first case, and with `left` and `right` in the second. A unary operator can be a boolean negation (`Not`), or the temporal operators `X`, `F` and `G`. The element `X` represents the *next* LTL operator, which requires its subformula to hold in the next state. `F` and `G` represent the temporal operators *eventually*—the subformula should hold at some point in the future—and *globally*—the subformula must always hold, from the current state on.

The binary elements can be boolean operators, like `And`, `Or` and `Implication`, or the temporal operators `U` and `R`. `U` represents the binary operator *until*, which captures that the left subformula must hold up to a point of transition where the right subformula holds. And the operator `R` represents the temporal operator *release*, similar to *until*, but with the additional requirement that the left and right subformulas should hold simultaneously at the moment of the transition.

All the operators are used to connect *atomic propositions*—the most basic type of formulas—which eventually evaluate to true or false in a particular model. An atomic proposition which is not evaluated is represented by the `Atomic` element, and once it is evaluated, the result is represented with a `Boolean` value. This last element, as well as its type `DT`, are defined in the data type dimension, used to represent common types such as integer, boolean or string. For a more detailed description of the data type dimension and its usage, see Sect. 3.3.

Note that `Formula`, `Unary` and `Binary` have potency `@0`, meaning that we cannot create instances of them. In this case, we achieve a similar effect by using zero potency like by defining abstract classes in object-oriented programming. We could have likewise refined those concepts by means of typing, separating them from the most specific elements (`Not`, `And`, `X`, etc.) by using two models

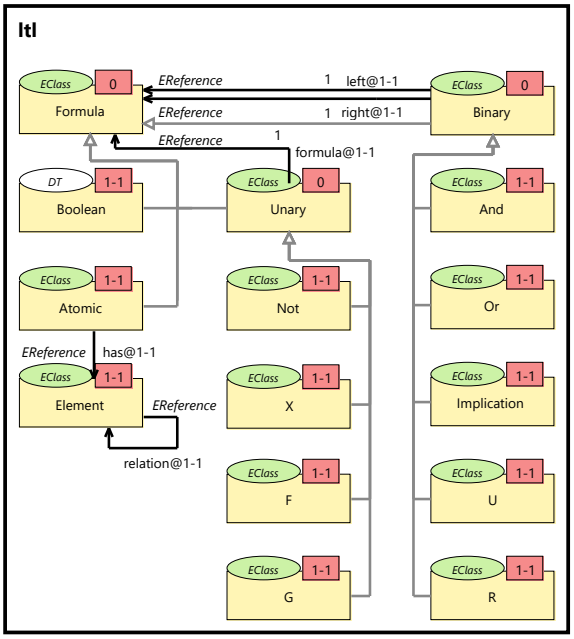


Figure 8: Supplementary LTL model

in two different levels—in a similar fashion as the `robolang` and `legolang` in Sect. 4.1. However, we wanted to illustrate that, thanks to the reuse of Ecore as the root of our hierarchies, we can define inheritance relations between nodes. In our approach, we do not enforce a particular way—inheritance/specialisation or metalevels/-classification (see Borgida et al. 1984; Kühne 2009; Mylopoulos et al. 1980)—of designing models and hierarchies with our framework; we provide the flexibility and leave this choice to the model designer. Moreover, this way of modelling makes it easier to read `ltl` as the model version of the EBNF grammar of the LTL language. This way, all operators define non-terminal symbols, `Boolean` and `Atomic` define terminals, and the relations define the structure of the symbols.

In Fig. 9, we display a specific property, called `property_1`, created as an instance of `robolang`—application dimension —, but also using types from `ltl`—supplementary dimension—to build a temporal property. The model encodes a consistency condition on the behaviour of the robot: when approaching an obstacle, backing away from it should clear the sensor-reading again. If

that is not the case, the robot's assumption about the environment—obstacles should not move—is incorrect, and the property is violated. This property, visualized in LTL syntax, is written as $G(obs \rightarrow X(\neg obs\, U\, to))$.

The model `property_1` also shows an example of double typing, that illustrates the way in which we use our framework to connect temporal properties to behavioural models. First, the property as a model is part of an application hierarchy, and is modelled using types from the supplementary dimension. And secondly, the atomic propositions get evaluated by matching them to the models representing executions of the behaviour, making use of the behavioural semantics for LTL, defined as MTs in Sect. 4.2. To simplify the visualization, if one of the types of a double-typed element belongs to Ecore, we omit it to increase readability.

For the two application types used in Fig. 9, we introduce a modifier in front of the type annotation. The meaning of `*Obstacle` and `*Timeout` is that `o` and `t` are not typed by them *directly*. The star prefix can be then interpreted as indirect typing or transitive typing, meaning that instead of having `o:Obstacle`, we may have `o:O1` and `O1:Obstacle`, or any number of intermediate types (like `O1`) in between. This notation increases the flexibility of our framework by allowing us to reason about elements in a very abstract way, without tying the notation to a model which is too specific to our purposes. For example, `property_1` is just concerned about obstacles, not particular instances of them. The differences on the different notations for types is also explained in more detail in Sect. 4.

Finally, one could even use LTL, or some other language, to define constraints in any of the levels of the hierarchy, exploiting the multilevel capabilities of the framework. The actual implementation of the MULTECORE tool can be instrumented to apply the language's semantics to evaluate such constraints against the models, without requiring any additional modelling mechanisms. For instance, just by using the non-temporal operators of LTL, it is possible to define propositional constraints like the one shown in Fig. 10; in fact, the

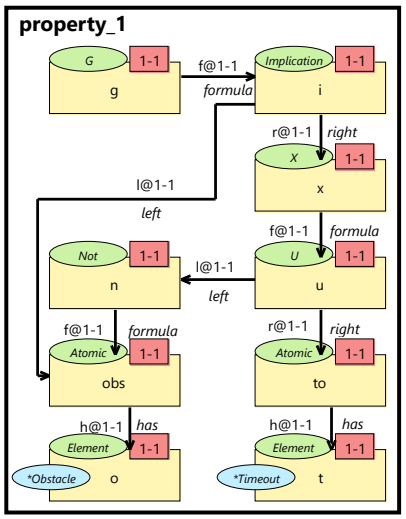


Figure 9: A temporal property with supplementary single-typed and double-typed elements

boolean logic language can be seen as a meta-language for temporal logics like LTL, and could be separated into different levels of a multilevel hierarchy, which we do not show here for the sake of brevity.

Let us assume that we want to define a constraint that forces instances of `GoForward` to only be connected to transitions triggered by an input of type `Obstacle`. This constraint, called `property_2`, is expressed by means of an implication, where the left-hand side contains the instances of `GoForward` and `Transition`, and the right-hand side reuses the same pattern—actually, the same elements—and adds the required instance of `Obstacle`, as shown in Fig. 10. The textual version of this constraint is just the implication $GFandT \rightarrow GFandTwithO$, where $GFandT$ and $GFandTwithO$ are the names of small models representing the atomic propositions.

3.3 Data type dimension

This dimension contains a single and fixed hierarchy of four levels, and defines the basic and most common data types used in most modelling tools, such as integers, strings and booleans. As with the two dimensions aforementioned, we choose Ecore for the root of the hierarchy for implementation

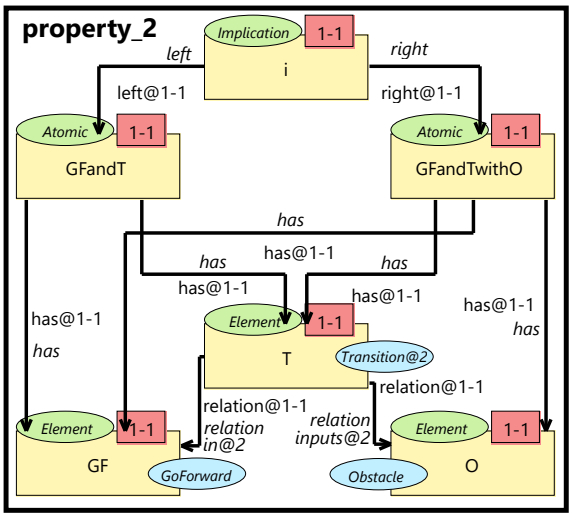


Figure 10: Example constraint using multilevel typing

reasons, but do not display it. By being orthogonal to the two aforementioned dimensions, data types are available to use in any other hierarchy. Figure 11 shows an excerpt of the three bottom levels of this hierarchy.

The top level displayed on Fig. 11 defines the concept of data type (`DT`), combined data type (`DT,DT`) and initial element (`Init`). The arrows represent operations, or definitions of them, between the different data types. For example, `Init` is used to define constants for some data types, hence the arrow `const`. The arrow `unop` indicate that an unary operation can relate two specific instances of `DT`. Similarly, `binop` represents a binary operation from a combination of two data types (`DT,DT`) to a single one. The two arrows `param1` and `param2` indicate how a combined data type is constructed as a combination of two single ones. The element `Attribute` can be used to double-type a node in any other hierarchy, indicating that it is an attribute. The `value` relation indicates the connection of such a node with its value.

In the second level, three data types are defined (`Boolean`, `Int` and `String`) as well as two of their combined types (`Boolean,Boolean` and `Int,String`). These combined data types can be used to define, for example, the binary operation `append` that takes a string and generates a new one by attaching a given integer to its end, and the `and` operation for boolean values. Note that all operations are defined as arrows between two nodes, but their semantics are specified using model transformations (see Sect. 4.3). The arrows from `I` are used to define the constants `zero` for integers and `empty` for strings. Finally, the unary operation `succ` is defined to get the successor of an integer.

The last level of the hierarchy is a graph with a countable infinite number of nodes and arrows, so we just display a few examples for illustration. The instances of the morphisms defined in the level above are named with the first letter of their respective types.

These graphs are mostly theoretical, and our proposal uses this hierarchy to establish a conceptual interface to the actual implementation of data types, in order to keep the formalisation separated from implementation particularities. Hence, this representation is consistent with the rest of the framework, including the definition of the semantics of the operations defined as morphisms, like `succ` for integers, as we show in Sect. 4. For the actual implementation (see Sect. 5), we ensure compatibility with the Ecore data types by reusing them, avoiding redefinition.

One example of the usage of data types has already appeared in Fig. 8. In the abstract syntax, an attribute is created by defining a new element whose type is not in its own hierarchy, but in the data type hierarchy. The framework forbids instantiation of types that are not defined in the data type hierarchy, so that the `Boolean` element is a valid instance of `DT`, and `true` is a valid instance of that attribute (i. e. its value) in the level below, where a specific LTL expression is defined and evaluated. As with the LTL syntax, this way of representing attributes would be cumbersome in practice. Nevertheless, a concrete syntax—graphical or textual—could be defined, such that attributes are represented in a more familiar way, e. g. as a list inside the node that contains the attributes.

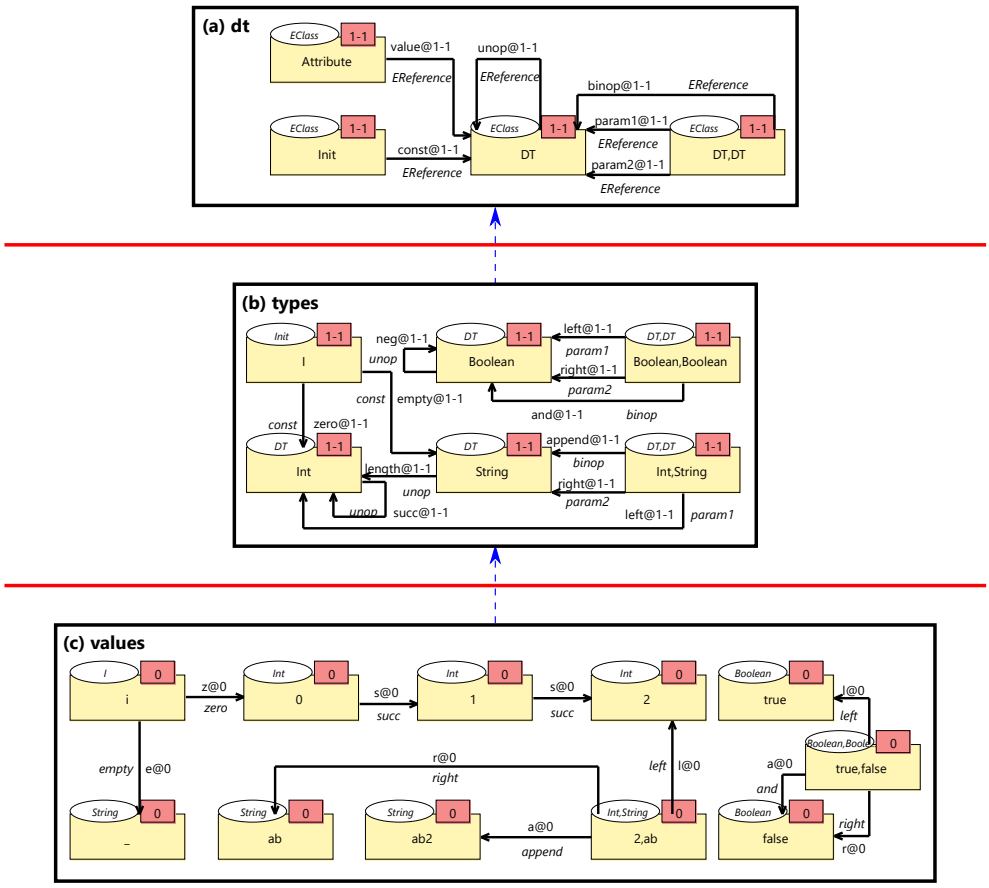


Figure 11: Fragment of the data type hierarchy

4 Defining Behaviour with Multilevel Coupled Model Transformations

The formal, flexible MLM approach introduced so far also gives us reusability when modelling behaviour. While modelling structure has advanced due to mature tools and frameworks, modelling behaviour has still a long way to go, especially because of the challenges related to the definition of their semantics. Moreover, understanding the behaviour of models is required to understand the behaviour of the derived software systems. Several approaches have been proposed for the definition and simulation of behaviour models based on model transformations (see for example Csertán et al. (2002), de Lara and Vangheluwe (2002), Rensink (2004), Rivera et al. (2009), Schürr and Rensink (2014) and Taentzer (2004)). Since most behaviour models have some commonality both in concepts and their semantics, reusing these model transformations across behaviour models would mean a significant gain. Hence, by using MLM in a metamodelling process for the definition of modelling languages we could exploit commonalities among these languages through abstraction, genericness and definition of behaviour by reusable model transformations.

In this section, we will build on our running example from the domain of robotics to explain our approach to reusable model transformations, namely, Multilevel Coupled Model Transformations (MCMTs). We will also compare MCMTs to other well-known approaches so that the advantages become clearer. To make this section easier to understand, we abstract away from the fact that a robot model could be defined several levels below the `robolang` metamodel, and say

that the robot model is defined by *a* `robolang` language.

As an example, we will now define the behavioural semantics of firing transitions. In a model specified using a `robolang` language, an indirect[1] instance of a transition connects two instances of (possibly the same) task together with a set of input-instances[2] . This can be seen in Fig. 7(c). In a running instance of a robot, such as Fig. 7(d), whenever we detect an instance of an input which is connected to a transition which in turn is connected to a 'source' task-instance, we can fire the transition and hence create the 'target' task-instance. By source and target task-instances of a transition we mean, respectively, task-instances which are connected by in and out relations to the transition. Using model transformation rules, we have now three options to define a rule for this behaviour.

Traditional two-level MT rule.
Without the usage of multilevel rules, that is, using two-level transformation rules, we would need to define one rule per instance of transition (see Figs. 12 and 13). Only for `robot_1` in Fig. 7(c) we already need seven transformation rules, one for each node of type `Transition`. And we would need more rules every time we add new structure to our model, or when we create another robot model, or when we extend the Robolang metamodel or any of its instances, e. g., a Robolang language for specifying Lego EV3 robots and their behaviour or a similar one for Arduino robots.

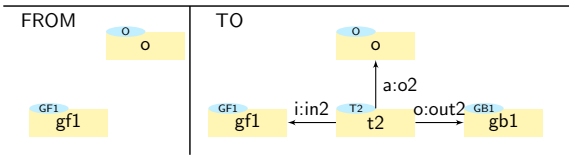


Figure 12: Behaviour for the transition `T2`

While this might do the job for one language, problems arise when considering reusability. First of all, the rules would be too specific and tied to the

[1] Hereafter we drop the word 'indirect'.
[2] We will use 'instance of input' and 'input-instance' interchangeably. The same is true for task and transition.

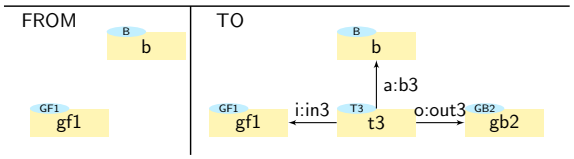


Figure 13: Behaviour for the transition `T3`

types `T2`, `T3`, `T4`, etc. This leads to proliferation in the sense that, as mentioned above, several similar rules must be defined. Hence, each transition instance would need a rule and each branch in the hierarchy would need its set of almost 'identical' rules.

The basic structure of these rules is outlined in Fig. 14. In its most general terms, a graph transformation rule is defined as a left L and a right R pattern (see Fig. 14). These patterns are graphs which are mapped to each other via graph morphisms l, r from—or into—a third graph I, such that L, R, I constitute either a span or a co-span, respectively (Ehrig et al. 2006, 2009; Mantz et al. 2015). In this paper we will use the co-span version, since it facilitates the moving of model elements without the two phases of delete-then-add. In order to apply a rule to a source graph S, first a match of the left pattern must be found in S, i. e. a graph homomorphism $m : L \to S$, then using a pushout construction followed by a pullback complement construction will create a target graph T. In case of several rules being applicable at the same time, there are different strategies to get rid of non-determinism in the literature of graph transformations and model transformations. Two examples are layering, i. e. prioritization of rules, and negative application conditions. This scenario is out of the scope of this paper, and will be considered in future extensions of the proposal.

Details of application conditions and theoretical results on graph transformations can be found in Ehrig et al. (2006). In addition, we omit in our sample rules most of the details introduced by the graph I, since it is automatically derived from L and R.

In typed transformation rules, the calculation of matches needs to fulfil a typing condition. That is, as seen in Fig. 14, all the triangles must be commutative. In other words, since both the rules and the models are typed by the same type graph, the rules are defined at the type graph level and are applied to typed graphs.

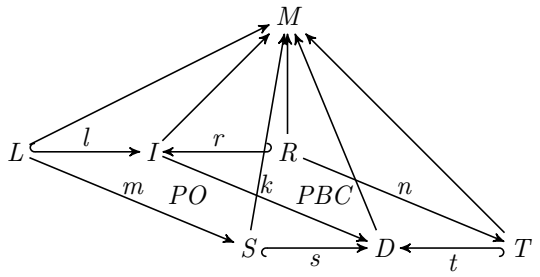


Figure 14: Typed MT rule

Multilevel MT rule.
Back to our example, multilevel rules enable us to define a single rule, by using the more abstract types `Task`, `Transition` and `Input` (see Fig. 15). This rule can be applied to all of the transitions `T2`, `T3`, etc, in Fig. 7(c), other robot models defined by the language in Fig. 7(b), as well as other models defined at lower levels with regard to the Robolang metamodel in Fig. 7(a).

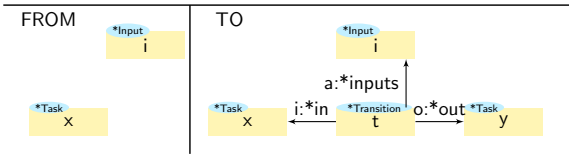


Figure 15: Abstract behaviour for firing transitions

While multilevel model transformations solve some problems, they introduce another problem related to case distinction. That is, the approach works fine in cases where the model on which the rules are applied—usually a running instance—contains the structure that is required by the rule, and, when all types required by the rule are existing in the same metalevel. If not, the rules will only be able to express behaviour in a generic way, but they will not be precise enough. In other words, the rules become too generic and imprecise: all transitions will fire (even if they are redefined in subsequent levels and given new behaviour) with the same conditions (i. e. detecting an input), and the rule would not take into account the type of tasks which the transition connects such that finding an input and a transition would lead to creating a random transition-instance with a random target task-instance. For example, in Fig. 7(c), an instance of `O` together with an instance of `TL` (which actually are not directly related) might trigger firing the transition `T5` and create an instance of `TR`. This is not the correct or desired behaviour for firing transitions.

The general structure of multilevel transformation rules is shown in Fig. 16. This could be considered as a method to relax the strictness of two-level model transformations through multilevel model transformations (see, for example, Atkinson et al. 2012, 2015c). This approach works only on multilevel metamodelling hierarchies (i. e., hierarchies which are not restricted to a fixed number of levels). To achieve flexibility, the rules can be defined over a type graph somewhere at a higher level in the hierarchy and applied to running instances at the bottom of the hierarchy (see Fig. 16). Types will be resolved by composing typing graph homomorphisms from the model on which the rule is applied and upwards to the level on which the type graph is defined.

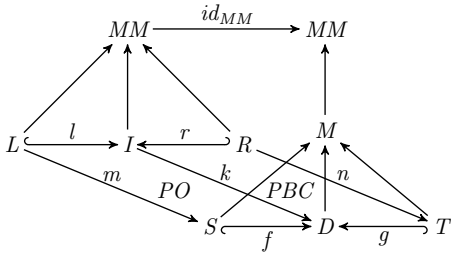


Figure 16: Multilevel MT rule

Multilevel Coupled MT rule.
We propose MCMT rules, as a means to overcome the issues of the two approaches aforementioned. Using MCMTs, the desired rule for firing transitions would be as shown in Fig. 17. In this graphical representation of the rules we show three

compartments: META, FROM and TO. Later in Sect. 5.2 we will show this rule in its textual syntax. This rule can be applied to fire any transition (except for the one connecting an initial task) in any robot model defined by the `legolang` language in Fig. 7(b). Similar to the multilevel rules, the variables `T, X, Y, I` could match several transition-instances, source and target task-instances, and input-instances, respectively. However, the difference here is that we would match each instance coupled with its type (hence the name 'coupled'). That is, when a variable is bound to a type, it will keep its type through the transformation. For example, consider applying the rule to the robot model in Fig. 7(c), if `I` is bound to `0`, then the only choice for binding `X` would be `GF`. This would lead to the binding of `T` to `T2` and `Y` to `GB1`. In this way, the right transition (`T2`) will be fired and the right target task-instance (`y:GB1`) will be created and connected to the right source task-instance (`x:GF`).

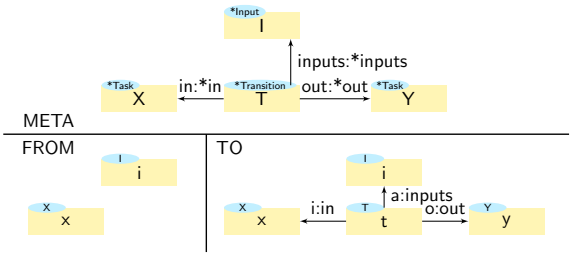


Figure 17: Rule FireTransition*: a transition gets fired by the associated input*

The general structure of an MCMT is displayed in Fig. 18. The figure can be visualized as two flat trees, each of them defined by typing chains and connected to each other by matching morphisms.

One of the trees contains the pattern that the user defines. It consists of the left and right parts of the rule (TO and FROM respectively), represented as L and R, and the interface I that contains the union of both L and R, hence the inclusion morphisms.[3]

These three graphs are typed by elements in the same typing chain, which is represented as a sequence of metamodels MM_x that ends with the root of the hierarchy tree MM_0 (Ecore in our case).

The other tree represents the actual hierarchy in which the rule is applied. Before applying the rule, we only have the graph S and its typing chain, consisting of type graphs TG_y. In order for the rule to be applied, it is required to find matches of all metamodel graphs MM_x into the actual hierarchy type graphs TG_y. These matches do not require to be 'parallel' in the sense that the difference of levels between the two sources of any two matching morphisms is not required to be equal to the difference of levels between the targets of those two morphisms. This is due to the flexibility in the specification of the number of levels that separate two metamodel graphs in the pattern. In terms of our example, if we add more intermediate levels to the Robolang hierarchy, and consequently the depth becomes bigger, the defined rules which were applicable before would still be applicable. Moreover, when the rules are defined with a flexible depth, they would fit or match different branches of the hierarchy, as long as $n < m$ (see MM_n and TG_m in Fig. 18). That is, the graph representing the pattern could be matched in several different ways to the same hierarchy, hence providing the flexibility that we require.

In both of the trees that define the application of a MCMT (see Fig. 18), and as already mentioned in Sect. 2.4, it is possible to define total morphisms from any graph on the tree to the root by composition of the partial morphisms that are depicted.

The match b_0 is trivial in the implementation since both MM_0 and TG_0 are Ecore. If all the metamodel graphs can be matched into the type graphs, such that every resulting square commutes, we proceed to find a match (homomorphisms) of the pattern graph L into the instance S. If this match (m) is successful, we construct the intermediate instance D by pushout, and then proceed to generate the actual target instance T by pullback complement.

[3] In all our scenarios, the interface I contains the left and right graphs since the morphisms l and r are monomorphisms, but this might not be the general case.

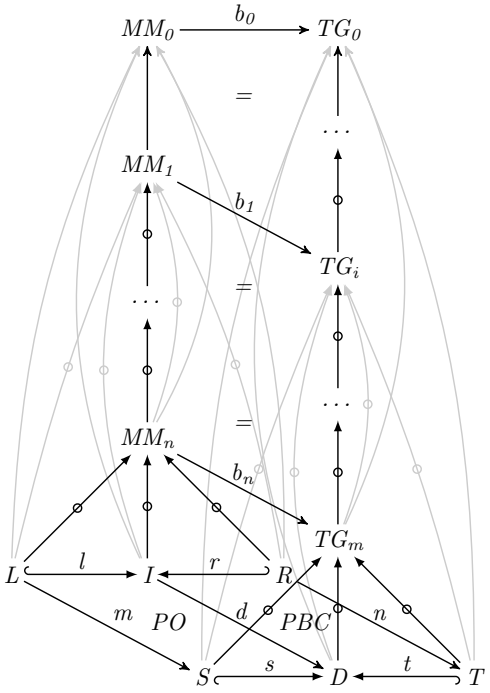


Figure 18: Diagram of a MCMT

To summarise, while existing approaches which employ reusable model transformations for the definition of behaviour models focus on traditional two-level modelling hierarchies and their affiliated two-level model transformations (see Kusel et al. 2015, 2013, for a survey), multilevel model transformations (Atkinson et al. 2012, 2015c) are relatively new and are not yet proven suitable for reuse and definition of model's behaviour. Our approach to multilevel model transformation builds on top of these original approaches to combine reusability with flexibility in the number of modelling levels. In other words, in the time of defining the rules, the height/depth of the modelling hierarchy (on which we intend to apply the rules) is not important since no matter how deep the hierarchy gets, the rules will still work, as long as $n < m$.

It is important to note that MCMTs can make use of three types of notation for the types. That is, we get three possibilities of specifying the type `t` of an element, with increasing degrees of genericness:

- With `t@2`—or any other integer value —, we indicate that the element `t` used as type is located two levels above. This notation has already been introduced in Sect. 3.1. Informally speaking, we could say that the type `t` can be found after a single 'jump' of fixed length (2, in this case).
- More generically, we can use `t@n` to leave open for the length of the jump. Using the same vocabulary as the previous point, this would represent one jump of *any* arbitrary length, which is useful when defining the patterns in MCMTs to increase flexibility. We do not, however, show any example of such notation in this paper.
- The `*t` syntax that was already introduced in Sect. 3.2, represents the concept of indirect typing. Again, we could say, informally, that this notation represents *any number of jumps* of *any* length. This `*`-notation together with its semantics provides the main corner-stone of our technique towards flexibility since it facilitates definition of rules without requiring to *a priori* decide on the depth of the modelling hierarchy.

4.1 MCMTs for application hierarchies

The definition of MCMT rules on application hierarchies are used to specify the behavioural semantics of the languages defined in those hierarchies. In this section, we show some examples of MCMT rules applied to the Robolang application hierarchy introduced in Sect. 3.1. This hierarchy and its rules are fully implemented as a proof-of-concept. The goal is to use MCMTs to describe complex behaviour of different robots at a higher level of abstraction, so that the rules can be executed, and thus simulate the behaviour of such robots.

Using the Robolang example, we have already explained the advantages of using MCMTs with respect to reusability and shortly presented a comparison to two-level and multilevel model transformation rules. Here we will show a few MCMTs which together define the behaviour of any robot which directly or indirectly conform to the `robolang` metamodel.

The first rule to consider is *Start* which is used to initialise the robot, that is, fire the first transition without any input (see Fig. 19). This rule is applicable as many times as existing instances of `Initial`. This rule says that, if in the model we have an initial task `Init:*Initial` which is connected to any task `X:*Task` by a transition `T:*Transition`, then in a running snapshot of that model if we find an instance `i:Init` we will construct the span with the instances `t:T` and `x:X`. Recall that `Init, T, X` are all variables, making this rule applicable to all initial tasks of all robot models (like the one in Fig. 7(c)) which are defined in an application hierarchy containing the `robolang` metamodel in a higher level in the hierarchy. Recall also that the notation `T:*Transition` denotes the fact that `T` might be an arbitrary number of levels below `Transition`.

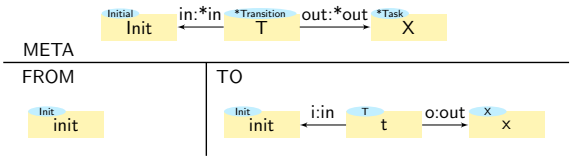


Figure 19: Rule Start*: firing initial transition*

After the initialisation, the *FireTransition* rule is used to fire ordinary transitions which rely on some input (see Fig. 17). Following the same reasoning as for the *Start* rule and as explained above, if in the model we have a transition `t:*Transition` (between two tasks `X:*Task` and `Y:*Task`) which has an input `I:*Input`, then if we have the instances `x:X` and `i:I` in a snapshot of the model, we will construct the fork with the instances `t:T` and `y:Y`.

The *DeleteTask* rule in Fig. 20 would delete the tasks and transitions which have already been fired and finished their actions.

The rules *InsertInput* and *InsertEffectiveInput* in Figs. 21 and 22 will, respectively, add a single input `i:I` to any snapshot and to a snapshot which already has an active task instance `x:X`.

Finally, the *DeleteInput* rule in Fig. 23 would delete an input `i:I` from any snapshot.

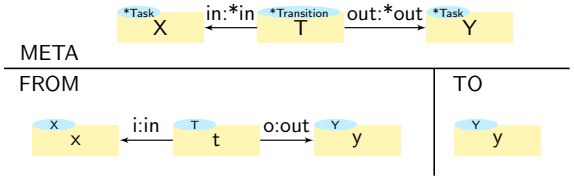


Figure 20: Rule DeleteTask*: a finished task is deleted*

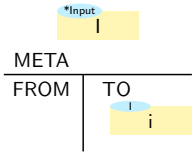


Figure 21: Rule InsertInput*: creates an input without constraints*

4.2 MCMTs for supplementary hierarchies

As examples in the supplementary hierarchy, we present a set of rules that illustrate the purpose of having such a hierarchy. Together with our LTL supplementary hierarchy, we will now show how to add execution semantics to the operators used in a temporal-logic specification. This semantics is applicable into any LTL property, since it does not refer to the particular application hierarchy that they are applied on. In other words, they would not change if we were to apply them in a different application hierarchy than Robolang.

We illustrate our encoding of the semantics of the temporal operators in the examples for the *next* and the *until* operators respectively. In the rule for *next* (see Fig. 24), the abstract notion of time advances, from the current state to the successor state, which for the evaluation of a formula $X\varphi$ to hold in the current state means that φ has to hold on the successor state. We achieve this by stripping off the corresponding node `x` from the current state of the evaluation of the property.

Notice that, in the rules for Robolang, we wrote variable names together with their types, e. g. `Init:Initial`, but in this rule for eliminating the next operator we write `X` and `Formula` without their types. We could have also written `X:EClass` and `Formula:EClass`, but this information is

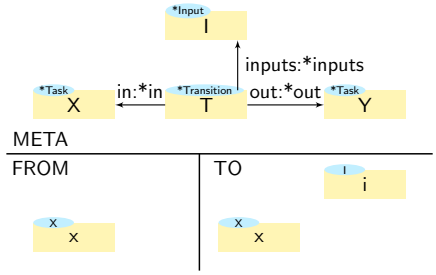


Figure 22: Rule InsertEffectiveInput*: creates an input that will cause the firing of a transition*

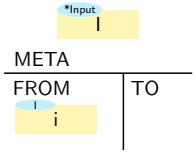


Figure 23: Rule DeleteInput*: deletes a previously existing input*

already given in the metamodel and hence unnecessary. Note that writing the element name `X` or `Formula` without its type is just a syntactic sugar to indicate that the element is a constant and not a variable. In such case, it must be matched to the exact element `X` or `Formula`, respectively.

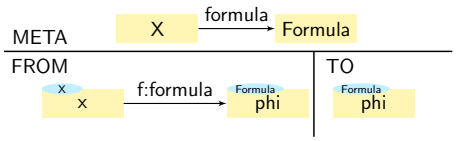


Figure 24: Eliminating the `X` *operator*

The rule for until (see Fig. 25) does not explicitly advance time. Rather it represents the well-known temporal equivalence that $\varphi\ U\ \psi = \psi \vee (\varphi \wedge X(\varphi\ U\ \psi))$.

The LTL example above also shows that coordination between MTs is necessary: the unrolling of the binary temporal operators is a pre-processing step, and the resulting expression is semantically equivalent to the original. As such, the same MTs could be applied again and again, only creating larger and larger subgraphs. It is essential that a kind of fairness condition between MTs is applied, so that the MTs for the atomic propositions get applied after at least one application of the

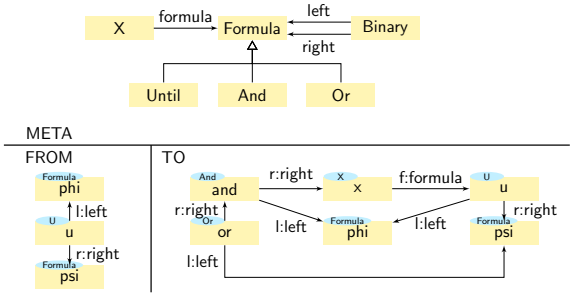


Figure 25: Unrolling the `U` *operator*

unrolling and shortcut evaluation of the complex terms to *true* or *false* according to the laws of the temporal logic. Furthermore, we have made sure that *every* temporal operator is unrolled *at least once*, since they can be nested. This can be achieved in different ways, either through defining MCMTs with explicit recursion (see e. g. Varró et al. 2007 for a graphical notation in VIATRA and Guerra and de Lara 2007 for a DPO approach with recursion), or through less explicit layering (see Chapter 12 in Ehrig et al. 2006).

4.3 MCMTs for the data type hierarchy

In this section, we briefly illustrate how the operations defined as arrows between data types can be provided with semantics by means of MCMTs.

In Fig. 26, we display the semantics of the `succ` operation defined for `Int`. The META part matches any element `X:Y` connected to `Int` by any arrow `Z:T`. For example, this could be applied to any node typed as an `Attribute`, with the arrow representing its declaration as an attribute of type integer. The rule matches (FROM) then any element `x:X` connected to a particular integer value, and switches that connection (TO) to the successor of that element. Hence, this MCMT provides the behaviour of the `successor_of` (i. e. increment) operation in a consistent manner with the rest of the framework, providing an interface from the actual implementation of the data types.

5 Implementation

The tools presented in this section provide the following features for our multilevel approach,

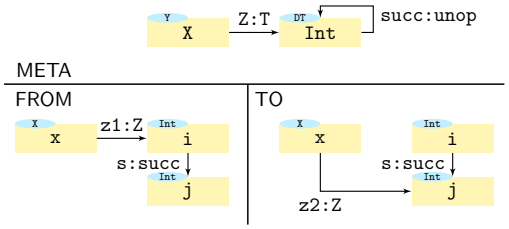


Figure 26: The succ *operation semantics as MCMT*

which we identified in Sects. 2, 3 and 4: (1) a graphical editor for modelling hierarchies that allows to define an unlimited number of levels, potencies for elements and make use of our three dimensions; (2) the idea of using a hierarchy as supplementary to an application one, keeping this one independent from the new aspects introduced into it; and (3) an editor for textual definition of MCMTs, which supports autocompletion and syntax highlighting.

5.1 MultEcore

This tool is a prototype which aims at combining the best from traditional (fixed-level) and multilevel modelling: the mature tool ecosystem and familiarity of the former, and the expressiveness and flexibility of the latter (Macías 2018; Macías et al. 2016, 2017). The first part is achieved by providing full compatibility and integration with EMF. Using MultEcore, model designers can seamlessly create a multilevel version of their hierarchies while still keeping all the advantages they get from fixed-level ones.

The tool provides the necessary functionalities to create multilevel hierarchies in EMF by allowing to get any two adjacent levels in the hierarchy represented as an Ecore metamodel and an XMI instance. This is achieved by storing a representation of each level in a way which is agnostic from any of those representations, and can be transformed into each of them on demand. Thus, all the already existing two-level tools designed for standard EMF can be used seamlessly with our hierarchies. Furthermore, this compatibility provides all the modelling capabilities of Ecore by default, such as abstract classes—although this can be also achieved with potency zero —, specialization and multiplicity for references and attributes.

The tool also provides a custom graphical editor implemented in Eclipse Sirius (The Eclipse Project 2016), designed specifically for multilevel modelling. This editor has been used to generate the example hierarchies displayed in Sect. 3. It provides a palette that is being currently adapted to dynamically include all available types (directly or via potency) from the models above it on the hierarchy. The hierarchies constructed in this manner are not fixed. Any level can be edited as a regular model, and new models can be added both at the bottom and in between existing ones. Note that Ecore, or the self-defining metamodel of choice, must always be located on top of the multilevel stack (see Sect. 2.2). So if the user wishes to add a model in top of the user-accessible hierarchy, she would still be adding an intermediate model between Ecore and the former top of the hierarchy. Figure 27 shows a screenshot of this editor, with the `robolang` language in the editing window.

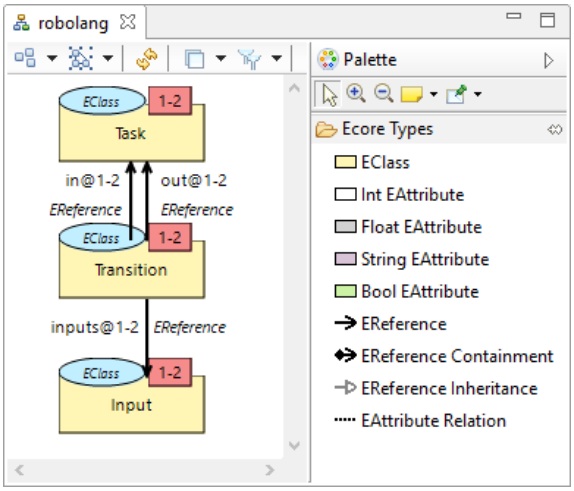


Figure 27: Screenshot from the tool, including the palette on the right-hand side

The specification of typing relations among elements in application and supplementary hierarchies is still not possible with the tool.

5.2 Textual DSL for MCMTs

For the specification of the transformation rules we need to decide how the rules should be defined

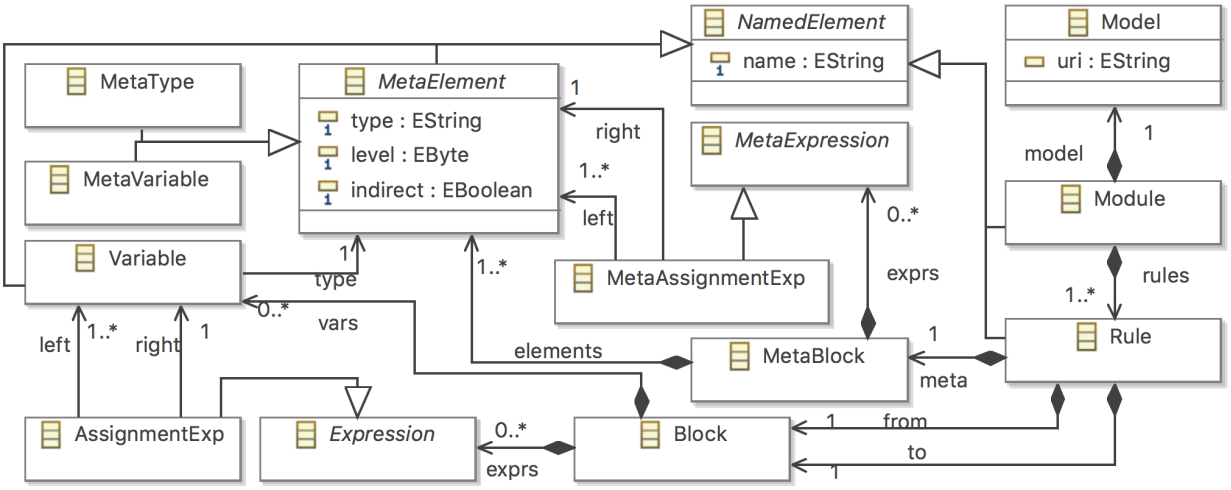


Figure 28: Excerpt of the metamodel for multilevel coupled transformation rules

and how they are applied. For this reason, we have created a textual Domain Specific Language (DSL) for the definition of multilevel coupled transformation rules, and designed an execution engine for simulation of instances by applying these rules.

The abstract syntax of this DSL is defined as an Ecore metamodel, of which an excerpt is partially shown in Fig. 28. Its primary element is Module, which is a collection of transformation rules defined over a model. This model URI may be passed as a parameter to every execution of those rules. This model is defined inside a multilevel hierarchy. The specific hierarchy to apply those transformation rules is automatically detected by the execution engine. Therefore those transformation rules are just applied to a single hierarchy in every single execution. Rule elements contain three organizational components, namely meta, from and to. All of them are blocks for graph pattern declaration. The meta block must contain a valid pattern, but from and to blocks may be empty. In the meta block, a pattern is specified by means of MetaVariable and MetaType declarations, and assignments between those MetaVariables. For those declarations we can use types defined at different levels in the MLM hierarchy. Level index starts from the model at the root of the hierarchy. In other words, the root model is at level 0. Additionally, those declarations may indicate direct or indirect typing. With direct typing we state that an element is a direct descendent (child) of its type (typing model element), meanwhile indirect typing indicates that the type of an element is within its hierarchy but it is not an immediate ancestor. Finally, in from and to blocks we can define patterns according to the MetaVariables and MetaTypes that we have previously specified in the meta section.

The textual syntax (conforming to this metamodel) of the *FireTransition* rule in Fig. 17 is shown in Fig. 29. Figure 30 shows the same rule defined by our editor. In the meta section, metavariables are declared by specifying their type from the root model of the multilevel hierarchy. Syntactically, type levels are specified as an index between square brackets. For example, mm[0]!Task states that Task is a type defined in the model named Robolang. Likewise, metatypes (i. e., constants) are declared in a similar way but the symbol $ (not shown in this example) is prefixed to the model keyword. Additionally, indirect typing is declared by an * prefixing the type of a metavariable or metatype (not shown in this example).

```
1  rule FireTransition {
2    meta {
3      X: model[0]!Task
4      Y: model[0]!Task
5      I: model[0]!Input
6      T: model[0]!Transition
7      inputs: model[0]!Transition.inputs
8      in: model[0]!Transition.in
9      out: model[0]!Transition.out
10     [T.inputs = I]
11     [T.in = X]
12     [T.out = Y]
13   }
14
15   from {
16     x: X
17     i: I
18   }
19
20   to {
21     x: X
22     y: Y
23     i: I
24     t: T
25     in: in
26     out: out
27     inputs: inputs
28     [t.inputs = i]
29     [t.in = x]
30     [t.out = y]
31   }
32 }
```

Figure 29: Textual representation of rule SendPartOut

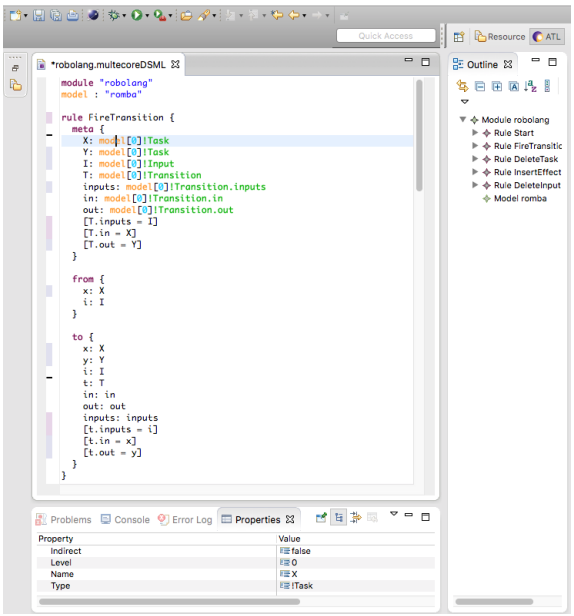


Figure 30: Textual editor for MCMTs

6 Related Work

In this section, we will identify two groups of works, one related to multilevel modelling and one related to reusable model transformations used for definition of behaviour in a multilevel setting.

First, we present some existing approaches that have tackled multilevel modelling by creating conceptual frameworks and tools. Some of the aspects discussed in the text are also synthesized in Tab. 1. The most widely used approach to the specification of multilevel frameworks is the usage of Orthogonal Classification Architecture (OCA), as defined in Atkinson and Kühne (2005). This architecture implies the definition of a linguistic metamodel that captures all features of the multilevel hierarchy. That is, the whole hierarchy becomes an instance of the linguistic metamodel. These approaches also exploit the concept of *clabject* (Atkinson 1997), which provides two facets for every modelling element, so that they can be seen as classes or as objects. Since clabjects stem from the traditional object-oriented programming, their realization into a metamodelling framework requires the linguistic metamodel, that all the levels must share, and which contains it together with other elements such as *field* and *constraint*.

Atkinson and Kühne (2005), who coined the term OCA, are also the creators of Melanee (Atkinson et al. 2015a), a tool for MLM. This tool has been developed with a stronger focus on editing capabilities (Atkinson and Gerbig 2016), as well as possible applications into the domains of executable models (Atkinson et al. 2015b). This tool allows for the specification of multilevel hierarchies, with potency features like durability and mutability for attributes and relations, which offer control to the level in which attributes are instantiated. Such fine tuning cannot be applied to classes, however, where only traditional potency is allowed. Multiple typing is also not supported in Melanee.

METADEPTH (de Lara and Guerra 2010a) is a proof-of-concept tool for a multilevel modelling

framework that requires the aforementioned linguistic metamodel. METADEPTH supports several interesting features such as model transformation reuse (Juan de Lara et al. 2015) and generic metamodelling (de Lara and Guerra 2010b). They support the concepts of potency formally defined in Rossini et al. (2014).

The formal definitions aforementioned are also used in the DPF workbench (Lamo et al. 2013), where the authors tackle multilevel modelling by means of graph morphisms and formal predicates. Some of these ideas have influenced the formalisation that we present. The use of potency is not supported by the DPF workbench, but the approach includes standard and leap potency as part of their specification.

AToMPM (Syriani et al. 2013) is a modelling framework highly focused on offering cloud and web tools. Modelverse (Van Mierlo et al. 2014), which is based on AToMPM, offers multilevel modelling functionalities by implementing the concept of clabject and building a linguistic metamodel that includes a synthetic typing relation.

FMMLx (Frank 2014) is one of the few approaches not using OCA. They apply the concept of *golden braid* with a similar realisation to ours: a topmost metamodel that defines itself, that can be transitively instantiated as many times as required to create a multilevel hierarchy. For potency, they allow marking features as *intrinsic*, which requires the specification of the level where that feature can be instantiated.

OMLM (Igamberdiev et al. 2016) uses an OCA, and extends it with a realisation dimension, which maps the modelling elements to their implementation counterparts. They use the standard concept of potency as depth, but do not apply it to attributes, which always have potency 1, so it sticks to the standard features of MLM which are common from most proposals.

DeepTelos (Jeusfeld and Neumayr 2016), is an extension of Telos which allows to implement OCA by using *most general concepts*, but does not natively require it. Their proposal comes closer to ours in both the formal background and the independence from a linguistic metamodel, although the approach does require it in the case of potencies, which need a custom set of elements specified that represent the allowed instantiations.

Dual Deep Modelling (Neumayr et al. 2016) is based on the definition of parallel hierarchies, with different depths, where relations from one to the other can be established. They are still based on OCA and the use of potency as depth, similar to most of the approaches mentioned here.

In a similar manner, SLICER (Selway et al. 2017) defines a complex and powerful set of tools to represent different relations among clabjects and the way these are instantiated, but it can still be classified as OCA since they require a linguistic metamodel that defines these concepts and relations.

The framework built around the NMeta metamodel (Hinkel 2016) depicts it at the top of their orthogonal stack of models, but we believe that the fact that the metamodel defines concepts like *Model* and *ModelElement* which are then instantiated to create the actual stack implies that the approach is also based on the OCA principles. The approach does not have explicit levels or potencies and uses inheritance to separate concepts from different levels of abstraction, although the authors claim that they allow for unbounded levels.

Finally, in Mallet et al. (2010), the authors assume that the only way to do multilevel modelling is by using the clabject approach. Therefore, they apply the same ideas for their implementation of multilevel modelling.

The way we specify potency can also be compared with previous formalisations of the concept, like the one defined in Rossini et al. (2014). Table 1 contemplates three possible types of potency: the original idea of potency as depth (*traditional*), the *leap* addition, which allows the instantiation to jump across levels, but only once, and the *range* potency presented in this paper, which can be combined with the first one and encompasses the second. Additionally, for the sake of completeness, we also included in Tab. 1 whether the approaches support multiple typing or not. Note that, for the sake of simplicity, the table does not distinguish

if the other approaches also provide potencies for arrows—as in our approach—or just for nodes.

Notice that, unlike other approaches, our realization makes the declared potency of an element independent of the potency of its type. For example, the default value of the potency of the arrow `out` in `robot_1` could be changed to some other one without being affected by the potency `@1-2` of its type arrow (`out` in `robolang`). Also, this kind of potency allows for the realisation of abstract classes by declaring values `0-0`, similarly to Atkinson and Gerbig (2012). However, the usage of Ecore as topmost metamodel already provides the concept of abstract classes, allowing for a clearer definition of these, in a similar manner to other tools like MetaDepth. Even further, the two values that we use to define potency are compatible with the traditional understanding of potency as *depth*, meaning that a third value representing this concept could be added when specifying the potency of an element, without compromising the integrity of our approach. This equivalence between three-valued potency and other multilevel concepts (e. g. leap potency, mutability and durability) is outlined in Macías et al. (2017), together with its equivalence to the potency specified by some multilevel tools.

Second, we present some works on reusable model transformations. In Chechik et al. (2016), two perspectives on model transformation reuse are presented: programming language- and MDSE-based. For each perspective, the authors discuss two approaches: subtyping and mapping, and lifting and aggregating. In Strüber et al. (2015) a variability-based graph transformation approach is introduced to tackle the performance problems that are introduced by systems in which a substantial number of the rules are similar to each other. In Sen et al. (2012) an approach to reusable model transformations is presented, which is based on sub-typing an effective part of the existing source metamodel. That is, the metamodel of the models to be transformed is made a subtype of a pruned metamodel. In this way, the models can be transformed by the same transformation rules which were written for the source metamodel.

To achieve genericness, in de Lara and Guerra (2014) and Sánchez Cuadrado et al. (2011) the rules are typed over a 'generic' metamodel which is called *concept*. Then, any metamodel to which there exist an embedding from the concept-metamodel (called binding) can be used by composition for the type-check during matching. Thus, any model in a hierarchy can be typed by the concept metamodel and get the transformation rules for free. However, it is not always straight forward to define the embedding morphism from the concept metamodel to the metamodel. This might be because the metamodel has several structures which have the same behaviour leading to several bindings. This is solved by introducing syntax for the definition of multiplicity in which the concept metamodel can be written in a generic way, however, in the realisation of the concept, this is just syntactic sugar for the definition of multiple concept metamodels. Moreover, finding reasonable embeddings due to structure mismatches (or heterogeneity) might be a challenge. Adapters and concept inheritance could be seen as solution of this problem, as seen Sánchez Cuadrado et al. (2011).

The notion of *concept* from de Lara and Guerra (2014) is extended in Durán et al. (2015) to parametric models, where the parameters have both structure and behaviour. In this case we not only have a mechanism for the reutilisation of transformations, but mechanisms for the reutilisation and composition of models with behaviour. The difficulties for finding embeddings is however even more difficult, since rules in parameter models must also be mapped to the corresponding target models.

An existing approach to multilevel model transformation rules which could be suitable for the definition of behaviour models is described in Atkinson et al. (2012, 2015c). However, in the current implementation of this approach, only the instances which are at the lowest level in the metamodelling hierarchy will be transformed. For example if a modelling hierarchy contains four levels and a transformation is defined on the

Table 1: A comparison of multilevel features from different approaches

Approach	*Canonical Architecture*	*Traditional potency*	*Leap potency*	*Range potency*	*Multiple typing*
Melanee	OCA	✓	—	—	—
Modelverse (AToMPM)	OCA	✓	—	—	—
MetaDepth	OCA	✓	—	✓	✓
DPF	OCA	✓	✓	—	—
FMMLx	Self-defining top MM	✓	—	—	—
OMLM	OCA	✓	—	—	—
DeepTelos	OCA	—	—	—	✓
Dual Deep Modelling	OCA	✓	—	—	—
SLICER	OCA	✓	—	—	✓
NMeta	OCA	—	—	—	—
This work	Self-defining top MM	✓	✓	✓	✓

highest level, only the model elements on the lowest level are transformed into the target model. This hinders the definition of metamodels representing language behaviour at any level in the stack, that is, these metamodels need to be at the next to bottom level. In Juan de Lara et al. (2015), an approach to multilevel modelling hierarchy for description of DSML is proposed, which is quite similar to our approach, however, it does not take coupling of model transformations into account.

7 Conclusions and Future Work

We have presented a formal description of a general-purpose multilevel modelling framework and illustrated how to apply such concepts into practical scenarios of behavioural modelling. We have provided illustrative examples of the different features of the formal framework and showed our implementation in two tools—prototypes based on EMF. We have argued how the provided formalisation favours flexibility in several aspects of the framework, and compared it to the main existing approaches to multilevel modelling.

The formalisation via graphs allows us to apply techniques from graph and category theory such as graph homomorphism, pullback and pushout. These techniques avoid potential ambiguities on the formal specification of our framework and provide us with powerful tools to achieve flexible and reusable multilevel modelling in both aspects of modelling hierarchy definition and model transformation specification.

In addition, our tools allow us to test these ideas in a more realistic manner, and detect new features that are desirable for our approach, like the novel concept of range-like potency.

Our current work stops short of executing model-transformations: given a model and a MCMT, we can check whether the transformation is applicable, and obtain a transformed model through its application. Given a set of MCMTs however, it is clear that several transformations might be applicable simultaneously, as mentioned in Sect. 4. In the case when they apply to overlapping parts of the graph, a particular sequence of applying MTs may not produce a confluent result, i. e. different application orders lead to different overall results. In some cases, this might require techniques for coordinating the application of the MT rules in order to achieve confluence. Hence the next natural steps of this work comprise finishing the execution engine for MCMTs and including rule coordination strategies. Moreover, we plan to improve the capabilities of the MULTECORE tool and test the validity of our results in new case studies.

References

Aßmann U., Zschaler S., Wagner G. (2006) Ontologies, Meta-models, and the Model-Driven Paradigm. In: Calero C., Ruiz F., Piattini M. (eds.) Ontologies for Software Engineering and Software Technology. Springer, pp. 249–273

Atkinson C. (1997) Meta-modelling for distributed object environments. In: Enterprise Distributed Object Computing Workshop (EDOC'97). IEEE, pp. 90–101

Atkinson C., Gerbig R. (1st Jan. 2012) Melanie: Multi-level Modeling and Ontology Engineering Environment. In: Proc. of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. MW '12. ACM, Innsbruck, Austria, 7:1–7:2 published

Atkinson C., Gerbig R. (1st Jan. 2016) Flexible Deep Modeling with Melanee. In: Betz S., Reimer U. (eds.) Modellierung 2016, 2.-4. März 2016, Karlsruhe - Workshopband. Modellierung 2016 Vol. 255. Gesellschaft für Informatik, Bonn, pp. 117–122

Atkinson C., Gerbig R., Fritzsche M. (2015a) A multi-level approach to modeling language extension in the enterprise systems domain. In: Information Systems 54, pp. 289–307

Atkinson C., Gerbig R., Metzger N. (2015b) On the Execution of Deep Models. In: 1st Intl. Workshop on Executable Modeling. CEUR Workshop Proceedings Vol. 1560

Atkinson C., Gerbig R., Tunjic C. (2012) Towards Multi-level Aware Model Transformations. In: Hu Z., de Lara J. (eds.) Theory and Practice of Model Transformations - 5th Intl. Conf., ICMT 2012. LNCS Vol. 7307. Springer, pp. 208–223

Atkinson C., Gerbig R., Tunjic C. V. (2015c) Enhancing classic transformation languages to support multi-level modeling. In: Software & Systems Modeling 14(2), pp. 645–666

Atkinson C., Kühne T. (2001a) Processes and products in a multi-level metamodeling architecture. In: International Journal of Software Engineering and Knowledge Engineering 11(06), pp. 761–783

Atkinson C., Kühne T. (2001b) The Essence of Multilevel Metamodeling In: UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings Gogolla M., Kobryn C. (eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 19–33

Atkinson C., Kühne T. (Oct. 2002) Rearchitecting the UML Infrastructure. In: ACM Trans. Model. Comput. Simul. 12(4), pp. 290–321

Atkinson C., Kühne T. (2005) Concepts for comparing modeling tool architectures. In: International Conference on Model Driven Engineering Languages and Systems. Springer, pp. 398–413

Atkinson C., Kühne T. (2008) Reducing accidental complexity in domain models. In: Software & Systems Modeling 7(3), pp. 345–359

Banzi M. (2008) Getting Started with Arduino, Ill. Make Books - Imprint of: O'Reilly Media, Sebastopol, CA

Bézivin J., Lemesle R. (1997) Ontology-based layered semantics for precise OA&D modeling. In: Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques. Springer

Borgida A., Mylopoulos J., Wong H. K. T. (1984) Generalization/Specialization as a Basis for Software Specification In: On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages Brodie M. L., Mylopoulos J., Schmidt J. W. (eds.) Springer New York, New York, NY, pp. 87–117

Brambilla M., Cabot J., Wimmer M. (2012) Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers

Chechik M., Famelis M., Salay R., Strüber D. (2016) Perspectives of Model Transformation Reuse. In: Ábrahám E., Huisman M. (eds.) Integrated Formal Methods: 12th Intl. Conf., IFM 2016, Proceedings. LNCS Vol. 9681. Springer, pp. 28–44

Csertán G., Huszerl G., Majzik I., Pap Z., Pataricza A., Varró D. (2002) VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models. In: 17th IEEE Intl. Conf. on Automated Software Engineering (ASE 2002). IEEE Computer Society, pp. 267–270

Durán F., Moreno-Delgado A., Orejas F., Zschaler S. (2015) Amalgamation of domain specific languages with behaviour. In: J.LAMP 86(1), pp. 208–235

The Eclipse Project (Dec. 2016) Eclipse Sirius. http://www.eclipse.org/sirius/

Ehrig H., Ehrig K., Prange U., Taentzer G. (2006) Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer

Ehrig H., Hermann F., Prange U. (2009) Cospan DPO Approach: An Alternative for DPO Graph Transformations. In: Bulletin of the EATCS 98, pp. 139–149

Ehrig H., Prange U., Taentzer G. (2004) Fundamental Theory for Typed Attributed Graph Transformation In: Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings Ehrig H., Engels G., Parisi-Presicce F., Rozenberg G. (eds.) Springer, pp. 161–177

Fowler M. (2011) Domain-Specific Languages. Addison-Wesley

Frank U. (2014) Multilevel Modeling - Toward a New Paradigm of Conceptual Modeling and Information Systems Design. In: Business & Information Systems Engineering 6, pp. 319–337

Gerbig R., Atkinson C., de Lara J., Guerra E. (2016) A Feature-based Comparison of Melanee and Metadepth. In: Atkinson C., Grossmann G., Clark T. (eds.) Proc. 3rd Intl. Workshop on Multi-Level Modelling. CEUR Workshop Proceedings Vol. 1722. CEUR-WS.org, pp. 25–34

Guerra E., de Lara J. (2007) Adding Recursion to Graph Transformation. In: 6th Intl. Workshop on Graph Transformation and Visual Modeling Techniques. ECEASST Vol. 6

Hinkel G. (2016) NMF: A Modeling Framework for the. NET Platform. KIT

Igamberdiev M., Grossmann G., Selway M., Stumptner M. (2016) An integrated multi-level modeling approach for industrial-scale data interoperability. In: Software & Systems Modeling, pp. 1–26

Jeusfeld M. A., Neumayr B. (2016) DeepTelos: Multi-level Modeling with Most General Instances. In: Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 35. Springer, pp. 198–211

Kelly S., Tolvanen J. (2008) Domain-Specific Modeling - Enabling Full Code Generation. Wiley http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

Kühne T. (2009) Contrasting Classification with Generalisation. In: Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling - Volume 96. APCCM '09. Australian Computer Society, Inc., Wellington, New Zealand, pp. 71–78

Kühne T., Schreiber D. (2007) Can programming be liberated from the two-level style: multi-level programming with DeepJava. In: ACM SIGPLAN Notices 42(10), pp. 229–244

Kusel A., Schönböck J., Wimmer M., Kappel G., Retschitzegger W., Schwinger W. (2015) Reuse in model-to-model transformation languages: are we there yet? In: Software & Systems Modeling 14(2), pp. 537–572

Kusel A., Schönböck J., Wimmer M., Retschitzegger W., Schwinger W., Kappel G. (2013) Reality Check for Model Transformation Reuse: The ATL Transformation Zoo Case Study. In: Baudry B., Dingel J., Lucio L., Vangheluwe H. (eds.) 2nd Workshop on the Analysis of Model Transformations (AMT 2013) Vol. 1077. CEUR-WS.org

Lamo Y., Wang X., Mantz F., Bech Ø., Sandven A., Rutle A. (2013) DPF Workbench: a multi-level language workbench for MDE.. In: Proceedings of the Estonian Academy of Sciences 62(1)

de Lara J., Guerra E., Cuadrado J. S. (Sept. 2015) A-posteriori typing for Model-Driven Engineering. In: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th Intl. Conf. on, pp. 156–165

de Lara J., Guerra E. (July 2010a) Deep meta-modelling with Metadepth. In: Objects, Models, Components, Patterns. LNCS Vol. 6141. Springer, pp. 1–20

de Lara J., Guerra E. (2010b) Generic Meta-modelling with Concepts, Templates and Mixin Layers. In: Petriu D. C., Rouquette N., Haugen Ø. (eds.) Model Driven Engineering Languages and Systems: 13th Intl. Conf., MODELS 2010, Proceedings, Part I. LNCS Vol. 6394. Springer, pp. 16–30

de Lara J., Guerra E. (2014) Towards the flexible reuse of model transformations: A formal approach based on graph transformation. In: J.LAMP 83(5-6), pp. 427–458

de Lara J., Guerra E., Sánchez Cuadrado J. (Dec. 2014) When and How to Use Multilevel Modelling. In: ACM Trans. Softw. Eng. Methodol. 24(2), 12:1–12:46

de Lara J., Guerra E., Sánchez Cuadrado J. (2015) Model-driven engineering with domain-specific meta-modelling languages. In: Software & Systems Modeling 14(1), pp. 429–459

de Lara J., Vangheluwe H. (2002) AToM^3: A Tool for Multi-formalism and Meta-modelling. In: Kutsche R., Weber H. (eds.) Fundamental Approaches to Software Engineering, FASE 2002. LNCS Vol. 2306. Springer, pp. 174–188

de Lara J., Vangheluwe H. (2010) Automating the transformation-based analysis of visual languages. In: Formal Aspects of Computing 22(3), pp. 297–326

Macías F. (2018) MultEcore Website http://ict.hvl.no/multecore/

Macías F., Guerra E., de Lara J. (2017) Towards rearchitecting meta-models into multi-level models. In: International Conference on Conceptual Modeling. Springer, pp. 59–68

Macías F., Rutle A., Stolz V. (2016) MultEcore: Combining The Best of Fixed-Level and Multi-level Metamodelling. In: 3rd International Workshop on Multi-Level Modelling (MULTI2016). CEUR Workshop Proceedings Vol. 1722

Macías F., Rutle A., Stolz V. (2017) Multi-level Modelling with MultEcore: A Contribution to the MULTI 2017 Challenge. In: 4th International Workshop on Multi-Level Modelling (MULTI2017). CEUR Workshop Proceedings Vol. 2019

Macías F., Scheffel T., Schmitz M., Wang R. (2016) Integration of Runtime Verification into Metamodeling for Simulation and Code Generation (Position Paper). In: Falcone Y., Sánchez C. (eds.) 16th Intl. Conf. Runtime Verification, RV 2016. LNCS Vol. 10012. Springer, pp. 454–461

Mallet F., Lagarde F., André C., Gérard S., Terrier F. (2010) An Automated Process for Implementing Multilevel Domain Models. In: van den Brand M., Gašević D., Gray J. (eds.) Software Language Engineering. LNCS Vol. 5969. Springer, pp. 314–333

Manna Z., Pnueli A. (1995) The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer

Mantz F., Taentzer G., Lamo Y., Wolter U. (2015) Co-evolving meta-models and their instance models: A formal approach based on graph transformation. In: Sci. Comput. Program. 104, pp. 2–43

Monk S. (2011) Programming Arduino Getting Started with Sketches, 1st. McGraw-Hill Education TAB

Mylopoulos J., Borgida A., Jarke M., Koubarakis M. (1990) Telos: Representing knowledge about information systems. In: ACM Transactions on Information Systems (TOIS) 8(4), pp. 325–362

Mylopoulos J., Feather M. S., Meyer B., Paolini P., Smith D. C. P., Hendrix G. G. (1980) Relationships Between and Among Models (discussion). In: Brodie M. L., Zilles S. N. (eds.) Proceedings of the Workshop on Data Abstraction, Databases and Conceptual Modelling, Pingree Park, Colorado, June 23-26, 1980. 2 Vol. 11. ACM Press, pp. 77–82

Neumayr B., Schuetz C. G., Jeusfeld M. A., Schrefl M. (2016) Dual deep modeling: multi-level modeling with dual potencies and its formalization in F-Logic. In: Software & Systems Modeling, pp. 1–36

Odell J. J. (1994) Power types. In: Journal of Object-Oriented Programming 7(2), p. 8

Odell J. J. (1998) Advanced object-oriented analysis and design using UML Vol. 12. Cambridge University Press

Pfaltz J. L., Nagl M., Böhlen B. (eds.) AGTIVE 2003. LNCS Vol. 3062. Springer

Rensink A. (2004) The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz J. L., Nagl M., Böhlen B. (eds.) AGTIVE 2003. LNCS Vol. 3062. Springer, pp. 479–485

Rivera J. E., Durán F., Vallecillo A. (2009) A graphical approach for modeling time-dependent behavior of DSLs. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009. IEEE Computer Society, pp. 51–55

Rossini A., de Lara J., Guerra E., Rutle A., Wolter U. (2014) A formalisation of deep metamodelling. In: Formal Aspects of Computing 26(6), pp. 1115–1152

Rutle A., Rossini A., Lamo Y., Wolter U. (2012) A formal approach to the specification and transformation of constraints in MDE. In: J. Log. Algebr. Program. 81(4), pp. 422–457

Sánchez Cuadrado J., Guerra E., de Lara J. (2011) Generic Model Transformations: Write Once, Reuse Everywhere. In: Cabot J., Visser E. (eds.) ICMT. LNCS Vol. 6707. Springer, pp. 62–77

Schürr A., Rensink A. (2014) Software and systems modeling with graph transformations. In: Software & Systems Modeling 13(1), pp. 171–172

Selway M., Stumptner M., Mayer W., Jordan A., Grossmann G., Schrefl M. (2017) A conceptual framework for large-scale ecosystem interoperability and industrial product lifecycles. In: Data & Knowledge Engineering 109, pp. 85–111

Sen S., Moha N., Mahé V., Barais O., Baudry B., Jézéquel J.-M. (2012) Reusable model transformations. In: Software & Systems Modeling 11(1), pp. 111–125

Steinberg D., Budinsky F., Paternostro M., Merks E. (2008) EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Professional

Strüber D., Rubin J., Chechik M., Taentzer G. (2015) A Variability-Based Approach to Reusable and Efficient Model Transformations. In: Egyed A., Schaefer I. (eds.) Fundamental Approaches to Software Engineering. LNCS Vol. 9033. Springer, pp. 283–298

Syriani E., Vangheluwe H., Mannadiar R., Hansen C., Van Mierlo S., Ergin H. (2013) AToMPM: A Web-based Modeling Environment. In: Demos/-Posters/StudentResearch@ MoDELS

Taentzer G. (2004) AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz J. L., Nagl M., Böhlen B. (eds.) Applications of Graph Transformations with Industrial Relevance, 2nd Intl. Workshop, AGTIVE 2003. LNCS Vol. 3062. Springer, pp. 446–453

Van Mierlo S., Barroca B., Vangheluwe H., Syriani E., Kühne T. (2014) Multi-level modelling in the Modelverse. In: MULTI@ MoDELS. CEUR Workshop Proceedings Vol. 1286

Varró G., Horváth Á., Varró D. (2007) Recursive Graph Pattern Matching. In: Third Intl. Symp. on Applications of Graph Transformations with Industrial Relevance, AGTIVE 2007. LNCS Vol. 5088. Springer, pp. 456–470