A Challenge Problem for the Verification and Validation of Model Transformations
Tamás Mészáros, Péter Fehér, Márk Asztalos & Tihamér Levendovszky

Abstract— Using patterns originating from the world of object-oriented software development such as design patterns, architectural patterns, and refactoring idioms has considerably simplified the design process of software systems. With the proliferation of Domain-Specific Languages, the generalization of OO patterns is a natural demand. A straightforward idea is to adapt OO patterns with automated tool support to the practice of Domain-Specific Modeling as well. A possible solution for that is using graph transformations to formalize and realize such patterns. One may expect, however, that the patterns are realized such that they are correct and executed exactly as expected. Thus, the verification and validation of the transformations is required. In this paper, we present how one can precisely define the requirements for a model transformation, and how to verify the requirements on the implemented transformations by hand. Since there are no existing solutions at the moment to perform these operations automatically, we provide the presented examples as practical problems to be solved to researchers developing computer-aided verification and validation systems.

1. Introduction
The paradigm of Model-Driven Engineering (MDE) advocates the use of models as the primary representation of information. Processing these models is realized by model transformations. This fact places a special emphasis on the provable properties of these transformations. The concept is simple, however, the transformations themselves may be very complex. Thus, the verification and validation of model transformations is very challenging. By verification we mean to answer the question “Has the system been developed well?” while by validation we mean to check “Has the expected system been developed?” However, in many cases, these two aspects highly overlap.

We found the verification and validation of model transformations necessary when we developed a special kind of model transformation that targets the automation of often recurring editing operations for arbitrary domains. In this paper, we present a case study about two of these transformations, show how we can formalize our expectations, and also provide a non-automated, manual way to prove that the transformations fulfill the requirements. Although there are research efforts that address the verification of transformations targeting one special problem domain, there is no generally applicable approach at the moment. We provide a non-automated solution here, however, we strongly believe that the questions that we raise and the properties that we prove may be a good challenge with practical relevance for further attempts to automation.

In the following two sections, we introduce the Active Model Pattern concept and its background. Then we present a pattern that unflattens a selected model fragment of state chart models. We show a possible way to formalize our expectations of the pattern. Then we present a realization of the pattern with graph transformation, and we show how we can prove its correctness (its correspondence to the specification). In Section 5, we present how we can realize static model patterns as an application of pattern framework; we formalize the expectations of the solution and also prove its correctness. In Section 6, a brief summary is provided about the state of the art of transformation verification, and discuss the possibilities of automating the verification process. Finally, we draw the conclusions and describe future research options.

2. Background
Using OO patterns such as design patterns [12], architectural patterns [4], and refactoring operations [11] has considerably simplified the design process of software systems. Patterns provide proven solutions for frequently recurring problems. They are usually described in an intuitive and informal way. For instance, many design patterns can be described as an incomplete UML model fragment that needs to be inserted into the destination class diagram. Code refactoring operations, however, are usually represented with example code fragments and textual explanations. With the proliferation of Domain-Specific Modeling Languages (DSMLs), the generalization of OO patterns is a natural demand. Nowadays, as
Domain-Specific Modeling (DSM) is gaining an increased popularity, a significant amount of experience and knowledge have been collected among the experts of the different domains. A straightforward idea is to adapt OO patterns with automated tool support to the practice of DSM as well. As DSMLs are meant to be used in arbitrary domains, thus, in accordance with [20], DSML patterns are referred to as **model patterns**.

The basics of the Active Model Pattern (AMP) concept were introduced in [19]. An AMP can be regarded as the combination of design patterns and refactoring operations in a domain-specific environment. The architecture of AMP is illustrated in Fig. 1.

<table>
<thead>
<tr>
<th>Static aspect</th>
<th>Operational aspect</th>
<th>Tracing aspect</th>
</tr>
</thead>
</table>

Fig. 1 The architecture of the Active Model Pattern Infrastructure

The AMP approach has three orthogonal aspects. The **static aspect** of AMP realizes the domain-specific, static model pattern support. Static model pattern is the common name of domain-specific design patterns, architectural patterns, and patterns in general with arbitrary intention. They can be considered incomplete model fragments that are inserted into the target model.

The AMP concept also includes universal design-time model manipulations, which are referred to as the **operational aspect** of AMP. Often recurring operation sequences during editing or model refactoring usually cannot be expressed with an incomplete model fragment with the static aspect. The operations can be considered on-demand, localized model transformations applied interactively. They can be realized either by a model transformation environment or by programming the modeling API directly (optionally supported with a proprietary DSL). The third aspect of AMP is the **tracing aspect**. It covers the detailed logging of model manipulations for certain operations, such as undo/redo purposes, or the recognition of the places where static patterns were applied to be able to update the instance model on changes to the pattern definition.

[19] proposes two different approaches to realize AMP support for arbitrary DSMLs: the one is based on the modification of the target metamodel, the other extends usual instance models with special flags. However, [19] focuses on the static aspect only.

### 3. Interactive Model Transformation

In the Visual Modeling and Transformation System (VMTS) [30] Transformation Framework (VTF) the model transformation environment consists of rewriting rules [8] that are chained by a control flow graph. In our approach, the Left Hand Side (LHS) and the Right Hand Side (RHS) of the rewriting rules are not separated. They are defined in the same transformation model and the role of the elements (i.e. matched or deleted for elements in the LHS; and created for the RHS) is specified with the **Action** attribute of the rule items: (i) an element $x$ of the LHS graph which element is not deleted in the RHS graph is marked as $x$.Action $= \text{match}$, (ii) elements created by the RHS graph are marked as $x$.Action $= \text{create}$ and (iii) elements deleted by the RHS graph are marked as $x$.Action $= \text{delete}$.

Formally, the topology of a $p = L \xleftarrow{i} K \xrightarrow{r} R$ rewriting rule in VMTS is defined by the $P^V$ pushout object [8] of $l$ and $r$:

$$K \xleftarrow{i} \xrightarrow{r} R$$

If we denote the range of an $x$ morphism as $\mathcal{R}_x$, then the elements matched by $\mathcal{R}_m \cap \mathcal{R}_a$ are not deleted. Elements matched by $\mathcal{R}_m \setminus \mathcal{R}_a$ are deleted and a subgraph isomorphic to subpattern $\mathcal{R}_a \setminus \mathcal{R}_m$ is created by the rule.

In VMTS, one can attach textual constraints to the rewriting rules as well. A graph rewriting can be applied if the match of the rule satisfies the constraints. The application of additional conditions is formalized in Definition 1 (typed, attributed graph rewriting and attributed type graphs are understood according to [8]).

**Definition 1** (typed, attributed graph rewriting with constraints). Given an attributed type graph $ATG$ with a data signature $DSIG$. A typed, attributed graph rewriting with constraints $p = L \xleftarrow{i} K \xrightarrow{r} R$, $\mathcal{X}$ consists of a typed, attributed graph rewriting and a set of constraints $\mathcal{X}$. Also, we have injective $l$, $r$.

A rewriting rule $p = L \xleftarrow{i} K \xrightarrow{r} R$, $\mathcal{X}$ is applicable to a typed, attributed graph $G$ via the match $m$ if there is a context graph $D$ such that (1) is a pushout and $m$ satisfies constraints $\mathcal{X}$, i.e. $m \in \mathcal{X}$.

In our implementation the set of constraints $\mathcal{X}$ is expressed using the Object Constraint Language (OCL, [25]). With OCL, we can describe both structural constraints and conditions on the attributes of the matched elements.

Rewriting rules in VMTS [2] do not use the abstract syntax of the host model but a custom DSML to be defined. The type of the model element created/matched by a rule element is defined by the **TargetTypes** attribute of the rule element. As it is allowed to specify multiple possible types for a single rule element to be matched, the **TargetTypes** attribute can hold more than one type identifier as well.

The complete semantics of the control flow model has been formalized in [23].

In [24], we have presented an interactive, graph rewriting-based model transformation environment that is...
capable of describing arbitrary static and operational patterns in a precise way. However, the formal verification aspect has not been discussed there which is the very topic of this paper. The framework is fully implemented in VMTS that serves as an execution environment for the patterns. By interactive we mean that the domain engineer executing the pattern application can visually trace and actively influence the transformation. The control flow can define any execution order of the rules, and it can instruct the system to ask for user input as well: to perform a branch in the control flow or to aid the matching phase of the rules. During the matching phase of the individual rules, the transformation can also be instructed to consider only those elements that are selected in the visual designer of the modeling environment. Thus, we can apply patterns in a localized manner. The semantics of interactive model transformations is described in [23].

Definition 2 (transformation parameters). The parameters of the transformation can be the following.
- Total or partial definition of the m morphisms for an arbitrary rewriting rule.
- Definition of the boolean functions evaluated at the decision points in the control flow.

Definition 3 (interactive model transformation) Interactive model transformation is a model transformation where the transformation parameters are not available at the beginning of the execution, but they are resolved on being used.

Since the universality of graph-rewriting is proven in [14], the framework is capable of describing any model pattern. However, when creating operational patterns for different purposes, we have recognized that model transformation definitions can become very complex depending on the pattern they realize. Because of the complexity, it is not always convenient that the transformation realizes exactly that pattern that it is intended to. Thus, the formal specification of the patterns as well as the formal analysis of the conformance of the realized transformation to the specification is required.

In the next section we show that (i) operational patterns can be realized with interactive model transformation such that they are easy to use and expressive enough in practical applications, (ii) it facilitates the formal verification of certain correctness conditions of the patterns.

4. Unflattening a State Chart Model

When editing state chart models, it is an often recurring operation (pattern) to reduce the size of the model by introducing composite states. The aim of the unflattening pattern is to realize this operation. The metamodel of our state chart domain is illustrated in Fig 2.

Fig. 2 Metamodel of the state chart language

States may be composite which is denoted by the containment edge on the State node. Each state has an Action property that defines the action to be performed when a specific state gets active. There are two special states as well: Start and Stop that express the entry and exit point of the state chart diagram. States are connected with Transitions. A transition also has an Action property that is applied if a transition is performed. The applicability of a transition is decided based on the Guard expression assigned to the transition. The outgoing transitions of a composite state behave as fallback transitions that are performed if a contained state is active, and neither of its outgoing transitions can be performed.

The application of the pattern is illustrated in Fig. 3. It wraps the selected states with a composite state, and optionally creates a start state inside the new composite state if exactly one selected element has incoming edges from outside the selection.

For each guard, action, target state triplet that is used for one transition for each selected state, a new transition is created with the same properties starting from the new composite state. Furthermore, the transitions with the same properties leaving the selection are removed.

A. Requirement specification

Firstly, we define an extended class of labeled graphs that is used to describe state chart models. Labeled graphs are understood according to [8].

Definition 4 (state chart graph) A state chart graph is a
G = (N, E, s, t, type, g, a) labeled graph where N and E denote the node and edge sets of the state chart, s: t: E → N denote the source and target functions. The type: (N ∪ E) → {state, start, stop, trans, cont} function defines the type of the graph elements: nodes can be states or start states or stop states while an edge may be either a transition or a containment edge. The g, a:E → string labels describe the related guard and action expressions for each transition-type edge. For edges of containment type the value of these labels is undefined.

Then we determine the application conditions of the pattern:

**Definition 5** (application conditions of the state chart unflattening pattern) To unflatten an $A_1 = (N_1, E_1, s_1, t_1, type_1, g_1)$ state chart model along an $L^\text{sel}_{N_1}$ state subset it is required that:

$$Y_1$$  
The selected states have a common container:

$$\exists e \in E_1, type_1(e) = \text{cont} \land s_1(e) = c \land t_1(e) = v.$$  

In the following, we enumerate those essential properties that ensure that the pattern realizes unflattening without changing the semantics of the state chart model.

**Definition 6** (correctness requirements of the state chart unflattening pattern) Assume two state chart graphs $A_1 = (N_1, E_1, s_1, t_1, type_1, g_1)$ and $A_2 = (N_2, E_2, s_2, t_2, type_2, g_2)$ according to Definition 4. Let $A_1$ denote the source graph describing the input state machine and $L^\text{sel}_{N_1}$ denotes the set of selected states in $A_1$, $A_2$ is the unflattened version of $A_1$ along $L^\text{sel}_{N_1}$ if we put the nodes defined by $L^\text{sel}_{N_1}$ into a hierarchy.

**φ₁** The pattern should preserve states, i.e. $\exists \text{map}_n: N_1 \rightarrow N_2$ injective mapping. Based on $\text{map}_n$ we define the image of the selected states in $N_2$ as $L^\text{sel}_{N_2} = \{v | v \in L^\text{sel}_{N_1}\}$

**φ₂** The pattern should preserve those transitions that connect either two selected or two not selected states. Namely $\forall e_1 \in E_1: type_1(e_1) = \text{trans} \land - \neg (s_1(e_1) \in L^\text{sel}_{N_1}) \Rightarrow \exists e_2 \in E_2: s_2(e_2) = \text{map}_n(s_1(e_1)) \land t_2(e_2) = \text{map}_n(t_1(e_1)) \land type_2(e_2) = \text{trans} \land g_2(e_2) = g_1(e_1) \land a_2(e_2) = a_1(e_1)\) $\bigoplus$ represents the exclusive or operation.

**(φ₃)** A new composite state should be created that contains all the selected states and has the common container of the selected states as its container: $\exists n \in N_2/\text{map}_n$:

(i) $\forall e \in L^\text{sel}_{N_1}: \exists e_1 \in E_2, type_2(e_1) = \text{cont} \land s_2(e_1) = n, t_2(e_1) = n$ and $\exists e_2 \notin E_2: type_2(e_2) = \text{cont} \land - \neg t_2(e_2) = n$

(ii) $\exists e_1 \in E_1, type_1(e_1) = \text{cont} \land - \neg t_1(e_1) = \text{map}_n(s_1(e_1)) \land t_2(e_1) = n_2$.

The later condition expresses that the previous container node of the selected nodes will be the container of the newly inserted container. ($\text{map}_n$ denotes the range of the $\text{map}_n$ function)

**(φ₄)** If for each selected state there is a transition leaving them towards the same target state, and all transitions have the same action and guard scripts, then these transitions should be removed and a new one should be created with the same guard, action expressions, and target state. Moreover, the new composite state should be its source state:

Let $T_{\text{common}} = \{e | e \in E \land type_1(e) = \text{trans} : (\forall v \in L^\text{sel}_{N_1}: \exists e' \in E \land s_1(e') = v \land type_1(e') = \text{trans} \land - \neg t_1(e') = t_1(e) \land g_1(e) = g_1(e) \land a_1(e) = a_1(e'))\}$

(i) $\forall e \in T_{\text{common}}: \exists e_2 \in E_2, type_2(e_2) = \text{trans} \land t_2(e_2) = \text{map}_n(t_1(e)) \land g_2(e_2) = g_1(e) \land a_2(e_2) = a_2(e) \land s_2(e_2) \in L^\text{sel}_{N_1}$ and

(ii) $\exists e_2, c \in E_2: type_2(e_2) = \text{trans} \land type_2(c) = \text{cont} \land t_2(e_2) = \text{map}_n(t_1(e)) \land a_2(e_2) = a_1(e) \land g_2(e_2) = g_1(e) \land t_2(c) = \text{map}_n(s_1(e)) \land s_2(e_2) = s_2(e) \land e_2$ is unique i.e. there is no other $e_2' \neq e_2$ with the same properties.

**(φ₅)** All the remaining transitions connecting a selected and an unselected state should be preserved:

$\forall e \in E_1 / T_{\text{common}} : s_1(e) \in L^\text{sel}_{N_1} \land \text{type}_1(e) = \text{trans} \Rightarrow \exists e' \in E_2: type_2(e') = \text{trans} \land s_2(e') = \text{map}_n(s_1(e)) \land t_2(e') = \text{map}_n(t_1(e)) \land a_2(e) = a_2(e') \land g_2(e) = g_2(e')$

**(φ₆)** All the remaining containment edges should be

---

**Fig. 4.** Control flow graph of the state chart unflattening transformation
preserved: \( \forall c \in E_t, \text{type}_e(c) = \text{cont} \land t_2(c) \in L_{N_2}^\text{rel} \Rightarrow \exists c' \in E_2: \text{type}_e(c') = \text{cont} \land t_2(c') = \text{map}_n(s_1(c)) \land t_2(c') = \text{map}_n(t_2(c)) \)

\((\varphi_2)\) \( A_2 \) does not contain any states or edges other than the ones referenced in points \( \varphi_1 \ldots \varphi_c \):

\[ |N_2| = |N_1| + 1 \quad (+1 \text{ represents the new composite state}) \]

\[ |E_2| = |E_1| + |T_{\text{common}}| \times (1/|L_{N_1}^\text{rel}| - 1) + 1 \]

\[ |T_{\text{common}}| \times (1/|L_{N_1}^\text{rel}| - 1) \text{ represents the removed transitions with common action and guard expressions and the added transitions for the new container state.} \]

\(+1 \text{ denotes the additional containment edge for the new container.}\)

Note that the constraints on the number of the nodes and edges together with the previous requirements imply the non-existence of any other nodes or edges in the unflattened model.

The following requirement is considered to be optional, it is not required, however, it can further optimize the state chart model:

\((\varphi_5)\) If there is exactly one state that has incoming transitions from unselected states among the selected states, then the target site of the incoming transitions can be set to the new composite state. A new start node should be created within the composite state and a single transition pointing from the new start state to the original target of the redirected transition. Namely, if we denote the only state with incoming edges from outside the selection as \( q_0 \) \( (q_0 \in L_{N_1}^\text{rel}, \exists e \in E_1: \text{type}_e(e) = \text{trans} \land s_1(e) \in L_{N_1}^\text{rel} \land t_1(e) = q_0 \) and \( \exists e' \in E_1: \text{type}_e(e') = \text{trans} \land s_1(e') \in L_{N_1}^\text{rel} \land t_1(e') \in L_{N_1}^\text{rel} / (q_0) \) and the new composite state as \( n_c \) \( (n_c \in N_2; 3c \in E_2 \land \text{type}_e(c) = \text{cont} \land s_2(c) = n_c \land t_2(c) \in L_{N_2}^\text{rel}) \) then

\[ \exists q_0 \Rightarrow \]

\[ (1)(3e_1 \in E_2: \text{type}_e(e_1) = \text{trans} \land s_2(e_1) \in L_{N_2}^\text{rel} \land t_2(e_1) \in (L_{N_2}^\text{rel})) \text{, and} \]

\[ (2) \forall e_1 \in E_1, s_1(e_1) \in L_{N_1}^\text{rel}, t_1(e_1) = q_0 \land \text{type}_e(e_1) = \text{trans} \Rightarrow \exists e_2 \in E_2: \text{type}_e(e_2) = \text{trans} \land s_2(e_2) = \text{map}_n(e_1) \land t_2(e_2) = n_c \land a_1(e_1) = a_2(e_2) \land g_1(e_1) = g_2(e_2), \text{ and} \]

\[ (3) \exists v \in N_2, c_2 \in E_2: \text{type}_e(v) = \text{start} \land \text{type}_e(c_2) = \text{trans} \land \text{type}_e(c_2) = \text{cont} \land s_2(c_2) = n_c \land t_2(c_2) = v \land s_2(e_2) = v \land t_2(e_2) = q_0 \land a_2(e_2) = \emptyset \land g_2(e_2) = \emptyset. \]

If an internal start state node is also created, then requirements \( \varphi_5 \) should be updated. \( \varphi_5 \) should be limited to the transitions leaving the selection except for \( T_{\text{common}}: \forall e \in E_1 / T_{\text{common}} / \{e'[t_2(e') \in L_{N_1}^\text{rel}) \}. \) In case of requirement \( \varphi_7 \) we have to consider an additional node (the internal start state) with a container edge and the transition connecting the start state to the selected elements.

\[ \text{B. Realization} \]

In the following, we present how we have realized the state chart unflattening operational pattern, using the interactive model transformation system. Fig. 4. illustrates the control flow graph of the transformation.

The transformation works as follows: the CountStart rule (Fig. 5(a)) matches all the states in the selection (innerNode) that have an incoming transition from a node outside the current selection, and puts those state nodes into a list. This is necessary, because if the list contains only one element (i.e. one selected node has one incoming edge), we may offer to create a start node inside the composite node, and redirect the incoming edges to the composite node (as shown in Fig. 3.).

The SelectOne rule (Fig. 5(c)) selects an arbitrary node (containedNode, called reference node from now on) within the selection, and places it into a newly created container (newContainer). Both the containedNode and the newContainerState nodes are flagged in order to be identified later. In addition, the container state is positioned and sized to cover the rectangle of the selection, and containedNode keeps its original absolute position inside it.

The SelectOne rule is left by two flow edges: the guard condition of the one (askStart) leading to CreateStart (Fig. 5(d)) asks the user whether to create an explicit start state within the new composite state. This question is asked only if CountStart found only one selected node with an incoming edge from outside the selection. If there is more than one such node or the user answers 'No', the execution proceeds to SelectNextTransition (Fig. 5(b)). Otherwise, CreateStart creates a new start state within the composite state and creates a transition within the start node and the state marked by CountStart. CreateStart is followed by the ReconnectStart (Fig. 5(e)) rule that sets the target node of each transition entering the selection to the new container state.

The SelectNextTransition, CheckTransitionsWithSame-Guard (Fig. 5(f)) and CreateFallbackEdge rules iterate through the edges of the reference node, and if the guard and action expression of an outgoing transition is used for one outgoing transition for each of the selected states, then that transition is pushed one level up to the composite state. This means that a transition can be pushed up without changing the semantics of the state chart, if all the selected states have an outgoing transition with the same guard expression. The SelectNextTransition (Fig. 5(b)) selects and marks one non-marked outgoing transition of the reference node, then CheckTransitionsWithSameGuard tries to match another selected state without an outgoing transition with the same guard and action expression (phrased in the textual constraints of the rule). If it fails, the transition can be pushed up (CreateFallbackEdge in Fig. 5(g)). Otherwise (and also after CreateFallbackEdge) a new transition is searched for by SelectNextTransition.

International Journal Publishers Group (IJPG) ©
moved into the composite state (SinkStates - Fig. 5(h)). Finally, the rule RemoveRedundantTransitions (Fig. 5(i)) removes those outgoing transitions of the selected nodes that have a corresponding outgoing transition on the composite node with the same guard and action expression and target state.

C. Correctness of the realized pattern

Proposition 1. The statechart unflattening transformation satisfies the requirements \( \varphi_1 \) – \( \varphi_8 \) provided that \( \gamma_1 \) applies to the \( L_{N_1}^{sel} \) selection.

Proof.

(\( \varphi_1 \)) No transformation rule deletes nodes or modifies their attributes.

(\( \varphi_2 \)) No transformation rule deletes an e edge or modifies the attributes of an e edge, if \( s_1(e) \in L_{N_1}^{sel} \Leftrightarrow t_1(e) \in L_{N_1}^{sel} \).

(\( \varphi_3 \)) The SelectOne rule is executed once, it creates a new state (newContainerState), and places it inside the previous container (oldContainer) of a state inside the selection. Based on \( \gamma_1 \), this is the common container of all the states of \( L_{N_1}^{sel} \). The SinkStates rule is executed once for each state inside oldContainer and moves them into newContainerState by deleting and recreating the containment edges.

(\( \varphi_4 \)) The requirement is ensured by the SelectNextTransition, CheckTransitionsWithSameGuard and CreateFallbackEdge rules. The SelectNextTransition rule matches a not marked outgoing transition (referenceTrans) of the reference node selected by SelectOne. The rule marks the selected transition to avoid matching it again by the same rule. The CheckTransitionsWithSameGuard rule matches referenceTrans again (between referenceNode and targetState), and tries to match another state (nodeWithoutSimilarTransition) from \( L_{N_1}^{sel} \) without a transition with the same labels and target state. If the match succeeds that implies that \( \exists v \in L_{N_1}^{sel}, \exists e \in E_1: t_1(e) = \text{trans} \land s_1(e) = v \land t_1(e) = t_3(r e f e r e n c e T r a n s) \land a_1(e) = a_3(\text{referenceTrans}) \land g_1(e) = g_3(\text{referenceTrans}) \). In this case SelectNextTransition is executed again. If CheckTransitionsWithSameGuard fails (i.e. \( \forall v \in L_{N_1}^{sel}, \exists e \in E_1: t_1(e) = \text{trans} \land s_1(e) = n \land t_1(e) = t_3(\text{referenceTrans}) \land a_1(e) = a_3(\text{referenceTrans}) \land g_1(e) = g_3(\text{referenceTrans}) \)), then CreateFallbackEdge is applied on referenceTrans. Consequently, \( \exists e' \in E_2: s_2(e') = \text{newContainerState} \land t_2(e') = \text{map}_0(t_3(\text{referenceTrans})) \). As SelectNextTransition and CheckTransitionsWithSameGuard select all the transitions of \( T_{\text{common}} \), and CreateFallbackEdge is applied on each of them, this applies to each

![Fig. 5. Rewriting rules to unflatten a state chart diagram](image-url)
5. Realizing Static Model Patterns with the Operational Aspect

Recall that static model patterns can be considered incomplete model fragments that can be inserted into other models. For example, Fig. 6 illustrates a static pattern for the state chart domain applied in case of user interaction [23]. The pattern presents an often recurring situation when an item should be highlighted if the mouse is positioned above it. BaseState can be considered the default state without highlighting, and if a MouseEnter event is received, a Highlight event is sent to the framework (in the action script for the edge), and the state machine proceeds to the MouseOver state.

In this section, we present how static patterns can be realized as the specialization of the operational aspect. We provide the formal description of the operation and prove the correctness of the solution.

A. Requirement specification

In the following, we show how to calculate the derived metatypes of a node. By derived metatypes of a node we mean a node type set: a node typed with either element of the set may replace the original node, provided that the same edges can be connected to it. We expect the set to be the smallest possible, but to cover all the possible types either directly or through inheritance (i.e. the instances of an A descendant B class are covered by A). Namely, the derived metatypes are distilled from the types of the connecting edges.

Definition 7 (derived metatypes) Having a typed graph \( G = (N, E, s, t, \text{type}) \) typed over graph with inheritance \( T = (G_T, I) \), the derived metatype of a node with an \( E' \subseteq E \) set of connecting edges is \( n_{\text{E'}}^E(n) = \{ n_T | n_T \in N_T \} \) where

\[
\bigcap_{(s, t) \in E'} \bigcap_{(s, t) \in E} \bigcap_{(s, t) \in E'} \bigcap_{(s, t) \in E}.\]

Where \( \text{clan}_n(n) \) means the set of nodes inherited from \( n \) and \( n \) itself.

We expect an operational pattern consisting of a single rewriting rule the following to realize static pattern insertion on execution:

Definition 8 (valid static model pattern rule). Assume that a static model pattern is defined by marking one or several elements of the host graph:

\[
\{ n | n \in N \} \cup \{ e | e \in E \} \cup \{ s | s \in S \} \cup \{ t | t \in T \}.
\]

We say that the VMTS rewriting rule model \( p = (N_p, E_p, S_p, T_p, \text{type}_p) \) realizes the static pattern defined by \( L^\text{set}_E \) if

\[
3 \text{ map}_E : L^\text{set}_E \rightarrow E_p \quad \text{and} \quad \text{map}_N : \{ n | n \in N \} \cup \{ e | e \in E \} \cup \{ s | s \in S \} \cup \{ t | t \in T \} \rightarrow N_p
\]

injective functions that satisfy the following requirements:

\[
\begin{align*}
(\forall s) \quad & n \in N^\text{set}_E : \text{map}_N(n).\text{TargetTypes} = \{ \text{type}(n) \} \land \text{map}_N(n).\text{Action} = 'create'. \\
(\forall e) \quad & e \in E^\text{set}_E : \text{map}_E(e).\text{TargetTypes} = \{ \text{type}(e) \} \land \text{map}_E(e).\text{Action} = 'create'. \\
(\forall s) \quad & n \in N : \text{map}_N(s).\text{TargetTypes} = N^\text{set}_T (n) \land \text{map}_N(n).\text{Action} = 'match'. \\
(\forall e) \quad & e \in E : \text{map}_E(s(e)) = \text{map}_E(t(e)) = \text{map}_E(e).
\end{align*}
\]
The requirements above describe that \((\varphi_1, \varphi_2)\) the rule elements related to selected elements should be marked as 'created' and their TargetType attribute should match the type of the related items; \(\varphi_i\) rule nodes related to not selected nodes connecting to selected edges should be marked as 'matched' and their target types should be the derived metatypes of the related node; \(\varphi_e\) expresses that the mapping is endpoint-preserving.

B. Semantic verification of the requirements, multi-typed graphs

The production rules used in VMTS allow multiple types to be matched for the same rule element as well as matching elements with a descendant type of the rule element. Consider the metamodel and instance model illustrated in Fig. 7.

![Fig. 7 Static pattern requiring multi-typed matching](image)

If we define a static pattern as illustrated in Fig. 7, b) without the c node, then it is clear that we may match nodes with type either C or D in place of the c node. The e and f edges can be connected to nodes with either of those types.

Thus, the Graph\(_\text{MTG}\) category is not able to describe the application of such a rule. This is why we introduce the multi-typed graphs (Graph\(_\text{MTG}\)) category that can handle both inheritance and multiple types for the same element. Using the multi-typed graphs formalism, we can interpret the TargetTypes attribute of the rewriting rule elements as if they were all the types of the corresponding element, and describe the execution of the rule as a double pushout.

Definition 9 (derived set function) Let \(\varphi: A \rightarrow B\) be a function. \(\pi_\varphi: P(A) \rightarrow P(B)\) denotes the set function which is interpreted as: \(\pi_\varphi(X) = \{\varphi(x)|x \in X\}, X \in P(A)\) (i.e., applies \(\varphi\) on each element of \(P(A)\), where \(P(A)\) means the powerset of \(A\)).

Definition 10 (multi-clan morphism) Given a type graph \(T = (G_T, I)\), and a graph \(H\), a multi-clan morphism type: \(H \rightarrow T\) consists of two functions \(\pi_{\text{G}_{\pi}}: N_H \rightarrow P(\pi_{\text{G}_{\pi}})\) \(\pi_{\text{E}_{\pi}}: E_H \rightarrow P(\pi_{\text{E}_{\pi}})\) such that (i) \(\forall e \in E_H: \text{type}_{\pi_{\pi}}(e) \subseteq \pi_{\text{clan}_1}(\pi_{\text{G}_{\pi}}(e))\), and (ii) \(\forall e \in E_H: \text{type}_{\pi_{\pi}}(e) \subseteq \pi_{\text{clan}_1}(\pi_{\text{G}_{\pi}}(e))\).

Definition 11 (Multi-instance graph or inheritance) Given a type graph (with inheritance) \(T\), a double \(G^* = (G, \text{type})\) of a graph \(G\) along with a multi-clan morphism \(\text{type}: G \rightarrow P(T)\) is called multi-instance of \(T\). \(G\) is also said to be multi-typed over \(T\).

Definition 12 (multi-typed graph morphism) Given a type graph \(T\), and multi-typed graphs \(G^*_1 = (G_1, \text{type}_1)\) and \(G^*_2 = (G_2, \text{type}_2)\), a multi-typed graph morphism \(f: G^*_1 \rightarrow G^*_2\) is a graph morphism \(f: G_1 \rightarrow G_2\) such that \(\forall l \in N_1 \cup E_1: \text{type}_{\pi_{\pi}}(f) \subseteq \pi_{\text{clan}_1}(\text{type}_{\pi_{\pi}}(f))\).

Theorem 1 (category Graph\(_\text{MTG}\)) Given a type graph \(T\), the category Graph\(_\text{MTG}\) of multi-typed graphs consists of multi-instance graphs of \(T\) as objects and multi-typed graph morphisms as morphisms.

Lemma 1 (relation of clans) Given a type graph \(T\), and two nodes \(n, n' \in N_T\) such that \(n' \in \text{clan}(n)\). In this case, \(\text{clan}(n') \subseteq \text{clan}(n)\).

Proof. Let \(n'' \in N_T\) be a node such that \(n'' \in \text{clan}(n')\). By definition, \(\exists\) path \(p_1\) from \(n''\) to \(n'\) through edges in \(I\), \(n'' \in \text{clan}(n') \Rightarrow \exists\) path \(p_2\) from \(n''\) to \(n'\) through edges in \(I\). Therefore, path \(p_1 + p_2\) is a path from \(n''\) to \(n'\). Since there is a path from \(n''\) to \(n'\), it means that \(n'' \in \text{clan}(n)\) by definition. Since \(\forall n'' \in N_T: n'' \in \text{clan}(n') \Rightarrow n'' \in \text{clan}(n)\), \(\text{clan}(n') \subseteq \text{clan}(n)\).

Lemma 2 (relation of clan sets) Given a type graph \(T\), and node sets \(N, N' \subseteq N_T\) such that \(N' \subseteq \bigcup \pi_{\text{clan}_1}(N)\). In this case, \(\bigcup \pi_{\text{clan}_1}(N') \subseteq \bigcup \pi_{\text{clan}_1}(N)\).

Proof. Follows from that of Lemma 1: for each \(n' \in N'\): \(n' \in \bigcup \pi_{\text{clan}_1}(N) \Rightarrow \exists\) \(n \in N\): \(n' \in \text{clan}(n)\). Thus, for each \(n' \in N'\): \(\exists\) \(n \in N\) so that \(\text{clan}(n') \subseteq \text{clan}(n)\). Consequently \(\forall n' \in N': \text{clan}(n') \subseteq \bigcup \pi_{\text{clan}_1}(N)\).

Lemma 3 (composition of multi-typed morphisms) Given a type graph with inheritance \(T\), three instance graphs \(A, B, C\). The composition of multi-typed morphisms \(f:A \rightarrow B\) and \(g:B \rightarrow C\) is a graph morphism \(g \circ f\: A \rightarrow C\). \(g \circ f\) is also multi-typed.

Proof. Let the type morphisms of \(A, B, C = \text{type}_{\pi_{\pi}}, \text{type}_{\pi_{\pi}}, \text{type}_{\pi_{\pi}}\), respectively.

\[g \circ f\] is a graph morphism, because it is composed by traditional graph morphisms by the composition operator, we need to prove that it satisfies the conditions of a multi-typed morphism, i.e.

\[\forall n \in N_A: \text{type}_{\pi_{\pi}}(g \circ f(n)) \subseteq \bigcup \pi_{\text{clan}_1}(\text{type}_{\pi_{\pi}}(n))\]

\[\forall e \in E_A: \text{type}_{\pi_{\pi}}(g \circ f(e)) \subseteq \bigcup \pi_{\text{clan}_1}(\text{type}_{\pi_{\pi}}(e))\]

\[\forall n \in N_C: \text{type}_{\pi_{\pi}}(g \circ f(n)) \subseteq \bigcup \pi_{\text{clan}_1}(\text{type}_{\pi_{\pi}}(n))\]

\[\forall e \in E_C: \text{type}_{\pi_{\pi}}(g \circ f(e)) \subseteq \bigcup \pi_{\text{clan}_1}(\text{type}_{\pi_{\pi}}(e))\]

Therefore:

\[\text{type}_{\pi_{\pi}}(g(f(e))) \subseteq \text{type}_{\pi_{\pi}}(e)\]

\[\text{type}_{\pi_{\pi}}(g(f(e))) \subseteq \text{type}_{\pi_{\pi}}(f(e))\]
Proof of Theorem 1. The composition of two multi-typed morphisms \( f: A \rightarrow B \) and \( g: B \rightarrow C \) is also a multi-typed morphism \( g \circ f: A \rightarrow C \) based on Lemma 3. The associativity law \( (h \circ g) \circ f = h \circ (g \circ f) \) for multi-typed morphisms is true, since it is true for ordinary graph morphisms. Similarly, the identity morphism \( \text{id}_A \) for each object \( A \) is the ordinary identity graph morphism of \( A \).

Having the GraphMTG category, we can prove that a rewriting rule fulfilling the requirements of a static pattern insertion rule really inserts a model fragment isomorphic to the pattern.

**Proposition 2.** Having a \( p \) rewriting rule realizing a static model pattern defined by \( \text{Selected} \) in the \( G \) host graph (Definition 8) typed over \( T \). Assume that we complete \( \text{Selected} \) to a valid subgraph \( \text{Selected}' \), i.e. for each \( n \in (N_G \setminus \text{Selected}) \land \exists e \in E_G \setminus \text{Selected} : s(e) = n \lor t(e) = n \) we add a new \( n' \) node typed with all the nodes of the \( T \) type graph and \( s(e) \) (or \( t(e) \)) to \( n' \). The execution of the \( p \) rule on the \( H \) host graph produces a model fragment that is topologically isomorphic to the extended pattern \( \text{Selected}' \) model pattern, provided that we allow injective matches only. Furthermore, the type of the newly created elements equals that of the corresponding elements in \( \text{Selected}' \).

Remark: The extension of \( \text{Selected} \) is necessary because isomorphism is defined on valid (sub)graphs only.

Proof. The execution of the \( p'=(L \leftarrow K \rightarrow R) \) rule is illustrated in Fig. 8 ((1) and (2) are pushouts). According to Definition 8, injective morphism \( \text{Selected}' \rightarrow R \) exists (note that the type of the extension nodes of \( \text{Selected}' \) is the superset of that of the corresponding elements in \( R \)). Thus \( \text{Selected} \rightarrow H' \) injective morphism exists as well.

(The existence of the morphism implies topological isomorphism and matching types for newly created elements.)

\[
\begin{align*}
G & \xrightarrow{\text{Selected}} \\
L & \xleftarrow{\text{Selected}} K \xrightarrow{(1)} R \\
H & \xleftarrow{\text{Selected}} D \xrightarrow{(2)} H'
\end{align*}
\]

Fig. 8 Execution of the rewriting rule inserting a static pattern

**C. Generating the static pattern insertion rule**

In the following, we present a transformation: by applying this transformation on the source model, we can define a mapping \( \text{map} \) that fulfils the requirements outlined in Definition 8.

The static pattern generator transformation (SPG) consists of three rules (illustrated in Fig. 9) that are executed exhaustively. The \( \text{rSelectedNode} \) rule matches all the selected nodes (\( \text{selectedNode} \)) once and creates the corresponding rule node (\( \text{ruleCreatedNode} \)) in the target model. The rule nodes are configured to perform the insertion (\( \text{Action} = 'create' \)) of a node with the appropriate type. Similarly, \( \text{rNotSelectedNode} \) matches all of those not selected nodes in the host model (\( \text{notSelectedNode} \)) that have a connecting selected edge. The rule creates a node in the target model with the derived metatype as target type, and configures the node to be matched during the execution (\( \text{Action} = 'match' \)). Similarly, \( \text{rSelectedEdge} \) matches all the selected edges once (\( \text{selectedEdge} \)), and creates the related rule edge (\( \text{ruleCreatedEdge} \)) in the target model. As \( \text{rSelectedNode} \) and \( \text{rNotSelectedNode} \) store the mapping between the elements of the source model and the elements of the target rewriting rule by creating a temporary reference between the corresponding nodes in the two different models, \( \text{rSelectedEdge} \) can identify the related endpoints to be connected in the target model. The metatype of the created edge matches that of the selected edge in the host model.

In the following, we verify each requirement defined in Definition 8 if and how the transformation satisfies them.

**Proposition 3.** Having a typed graph \( G=(N,E,s,t,type) \) typed over \( T \), applying the SPG transformation on the host graph \( G \) the resulting \( H \) graph satisfies requirement \( \varphi_1 \) of Definition 8.

Proof. With the single execution of the rule \( \text{CreatedNode} \), we match an \( n \in L_{\text{G}}^{\text{col}} \) node for node\( \text{Created} \) and create a corresponding \( n' \) rule element, for that \( n'.\text{TargetTypes} = \{\text{type}(n)\} \land n'.\text{Action} = 'create' \). The rule also creates a reference between \( n \) and \( n' \), which reference appears as a NAC on the left side of the rule. Thus, the same \( n \) node cannot be matched twice for node\( \text{Created} \), i.e. it cannot be modified during a repeated execution. The exhaustive execution of the rule terminates based on [32]: with the repetitive joining of the LHS graph and the RHS graph of the resulting rule based on the Concurrency Theorem, the number of elements of the resulting LHS graph will be strictly monotonic increasing. Consequently, the exhaustive execution processes each \( n \in L_{\text{G}}^{\text{col}} \) node, and creates a corresponding \( n' \) node with \( n'.\text{TargetTypes} = \{\text{type}(n)\} \land n'.\text{Action} = 'create' \).
Proposition 4. Having a typed graph \( G=(N,E,s,t,\text{type}) \) typed over \( T \), applying the SPG transformation on the host graph \( G \) the resulting \( H \) graph satisfies requirement \( \varphi_3 \).

Proof. The proof follows that of Proposition 3: rules MatchedNode and MatchedNode2 match non-marked nodes with marked incoming or outgoing edges and create a corresponding rule node (nodeMatchedRule) for each such node. The rules also set the TargetType attribute to \( N^\text{g}_E \{ \ell \} (n) \), and the Action attribute to 'match'.

Proposition 5. Having a typed graph \( G=(N,E,s,t,\text{type}) \) typed over \( T \), applying the SPG transformation on the host graph \( G \) the resulting \( H \) graph satisfies requirements \( \varphi_2 \) and \( \varphi_4 \).

Proof. Rule CreatedEdge matches each \( e \in E^\text{set} \) edge and creates a corresponding rule edge with the requested TargetType and Action attributes. Similarly to rules CreatedNode, MatchedNode and MatchedNode2, CreatedEdge is executed exhaustively and the already processed edges are excluded from the following match because of the reference created between the matched and the created rule edge. As \( \varphi_1 \) and \( \varphi_2 \) are satisfied by the other rewriting rules, matches for the \( p\text{NodeFrom} (= \text{map}_p(\text{nodeFrom})) \) and \( p\text{NodeTo} (= \text{map}_p(\text{nodeTo})) \) rule elements in the output model should exist for each match for nodeFrom and nodeTo. createdEdgeRule corresponds to \( \text{map}_E(\text{createdEdge}) \), thus \( \text{map}_p \) and \( \text{map}_E \) functions are edge preserving (\( \varphi_4 \)).

6. Related Work

In [31] a GME [18] solution is introduced: the Embedded Constraint Language (ECL) is based on OCL [25] but has imperative features as well. Using ECL, one can describe model refactoring imperatively. The transformation framework also supports localized execution (i.e., selecting those elements of the host model that are involved in the transformation), however, this approach is rather close to direct API programming.

AToM³ [5] provides a visual designer to build graph rewriting-based transformations, and facilitates the immediate visualization of the changes performed on the models. Furthermore, we can also access the UI through the API from the rules. However, AToM³ does not support localized transformations, and one cannot influence the pattern matching process directly.

AGG [26] also supports the visual specification of graph rewriting-based model transformations, furthermore, it facilitates the step-by-step execution of both the individual rules and the whole transformation. One can initialize or modify a match manually, and select the rules to execute. An AGG-based solution [3] applies graph rewriting to describe inplace model refactorings as well. As AGG has a layered, sequential control flow, it does not support user interactions and conditional branching.

[28] suggests graph rewriting-based model transformation for refactoring operations. The solution generates executable code from the visual specification of the rewriting rules. However, it lacks the possibility to model the execution order of the rules, thus, it needs to be written in plain JAVA.

Siikarla [27] introduces another approach, where the transformation steps are organized by an automatically generated task graph, the state of which can be persisted and reloaded. A task graph may contain automatable tasks and tasks that require human interaction as well. In this approach, we have less influence on the control flow of the transformation. It is similar to the layered approach that describes some sort of precedence of the rules.

There are several design pattern description languages. Most of them aim at their formal, and more precise description than it can be achieved with UML. A comparative discussion of formal pattern languages can be found in [9]. The primary target of these specification languages is limited to object-oriented design patterns only.


[29] proposes a formalization language, it rather extends the informal UML models. It offers first order logic to the static structure constraints, and temporal logic of actions (TLA). These models form a language referred to as balanced pattern specification language (BPSL).

[22] proposes a language called design pattern modeling language (DPML), which can be considered a DSML describing OO design patterns.

[15] describes a UML-based language, namely, the Role-Based Metamodeling Language (RBML), which is able to specify UML design patterns. This approach treats domain patterns as templates, where the parameters are roles. A tool generates models from this language.

The language RSL [10] is a formal language to specify design patterns. This language is used to specify OO design patterns. In this case, the common formal language is the root metamodel equipped with an instantiation relationship. The metamodel that can be derived across multiple levels, serves as an indirection that provides more flexibility and scalability in the pattern description language. The case studies and the tool support (DePMoVe) of this approach targets OO design patterns, while our approach facilitates arbitrary domain-specific design patterns.

The method proposed by [6] specifies design patterns in logic theorems using first-order logic, TLA, and Prolog. The formalization method targets OO design patterns only.

The work presented in [16] offers a finer-grained formalization of design patterns than that can be achieved with OCL. The proposed language is VDM++ with object calculus. The application of the method is not extended to arbitrary design patterns.

LePUS [7][13] is a formal language with tool support to describe and verify patterns and their applications. The application area of LePUS has been OO design patterns.
The tool support targets the Java programming language.

7. Discussion

Analysis of graph rewriting-based model transformations is an important research field, because model transformations are used extensively and usually in an automated way in model-based software development methods [34]. In this paper, offline analysis has been carried out, where offline means that only the very definition of the transformation and the language of the processed models are taken into account. Therefore, the result of the analysis is independent from the concrete input models, and, as in the case of the two transformations presented in this paper, the verification needs to be performed only once. The algorithmic verification of graph rewriting-based model transformation is undecidable in general [33], therefore, it would be beneficial to find certain subsets of transformations or subsets of the properties to be verified that have practical relevance and can still be efficiently analyzed.

Most often, formal verification of model transformations is performed manually or the methods can be applied only to a certain transformation class in specific application domains or for the analysis of only certain types of properties [36]. It is a usual approach that the verification of a model transformation is performed by converting the transformation itself into a general purpose formal domain where analysis methods are already available. A typical example is the translation of a transformation into the input of theorem provers, or a special analyzer tools [35]. The disadvantage of such methods is that during the analysis of a model transformation, we need to define a mapping, i.e. we translate the requirements to be verified from the current application domain of the transformation to the formal domain where automated analysis methods are available. The requirements can be complex and there may be domain-specific properties that are hard to be interpreted in another, more general domain. In the opposite direction, when an error is recognized in the general formal domain, it should be also interpreted in the application domain. It can be seen that, because the mapping is not always symmetric between the two domains, it is hard to provide the mapping of the domain specific properties and the found problems of the analysis domain.

Automated methods can be found for the analysis of rewriting rules [37]. For example, the properties $\phi_1$ and $\phi_2$ of the first transformation of this paper can be proved automatically, because only the definition of the individual rules needs to be examined. If one can show that no rule deletes states, the execution order of the rules is not relevant anymore from the point of view of preserving such elements. However, an important question in the verification of model transformations is the analysis of the imperative code. Usually, it is not easy to decide if these code fragments have side effects, and which attributes remain intact. The offline analysis becomes more difficult if complex control mechanisms should be taken into account. For example, in the case of $\phi_2$, multiple rules together guarantee the satisfaction of the specified property. Currently, there are no automated methods for the analysis of model transformations with such complex control mechanisms. However, other control mechanisms such as layering or explicit rule sequences are also an option, and, for example, for the analysis of the termination of such transformations, one can find more advanced methods [38].

8. Conclusion

Our work focuses on the formal verification of AMP operational patterns and the AMP static pattern realized as the specialization of the operational aspect. We have proposed graph rewriting-based interactive model transformations to formalize and describe the semantics of the patterns. Also, we have shown a way to formalize the requirements against a pattern realization using graph theory constructs. Furthermore, we have presented how the correspondence of the realization to the requirement specification can be shown. We have shown how operational patterns can be realized with interactive model transformations. It has been illustrated that they are easy to use and expressive enough in practical applications. As opposed to other representations, interactive graph transformations facilitate the formal verification of a wide variety of correctness conditions defined on the patterns. This is possible because of the strong mathematical background of graph transformations. The necessity and applicability of our approach is illustrated with an existing pattern unflattening a selected fragment of state chart models. We have formalized the expectations against the static aspect and proved that the proposed solution fulfills the requirements. Further fully developed case studies can be found in [23]. We have also shown how we can realize the static aspect of AMP as the application of the operational aspect. The presented formalization techniques can be utilized in case of general graph rewriting-based model transformations as well.

The model transformations presented in this paper provide well-defined case studies for offline analysis methods. We have shown examples for functional properties to be verified. It is also important how the verifiable properties can be specified, since a formal language is needed that is able to express the functional properties to be verified in the current application domain. Some of the properties presented in this paper can be verified by analyzing the individual rewriting rules separately. We have also provided examples for properties whose verification needs more complex examination of the control mechanism of the model transformations as a challenge problem for the future research in verification of model transformations.
Acknowledgment

This work was partially supported by the European Union and the European Social Fund through project FuturoICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

References


[34] Applications of graph transformations with industrial relevance. AGTIVE 2007 Graph Transformation Tool Contest, pages 487–492. Springer-Verlag, Berlin, Heidelberg.


Tamás Mészáros is a full-time senior lecturer at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics. His main research area is metamodeling, efficient model transformations, model patterns and model animation.

Péter Fehér is a PhD student at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics. His main research area is metamodeling, model transformation, pattern matching and the verification of model transformations.

Márk Asztalos is an assistant lecturer at the Budapest University of Technology and Economics, Department of Automation and Applied Informatics. His main research area is metamodeling, model transformation, and the offline verification of model transformations.

Tihamér Levendovszky Tihamér Levendovszky is a Research Assistant Professor at Vanderbilt University. He received his MSc and PhD degrees from Budapest University of Technology and Economics in 2000 and 2006. His interests are automated software engineering, model-based engineering, application of formal methods, and performance analysis of software systems.