

# Using Workflows to Automate Activities in MDE Tools

Miguel Andrés Gamboa and Eugene Syriani<sup>(✉)</sup>

Université de Montréal, Montreal, Canada  
{gamboagm,syriani}@iro.umontreal.ca

**Abstract.** Model-driven engineering (MDE) enables to generate software tools by systematically modeling and transforming these models. However, the usability of these tools is far from efficient. Common MDE activities, such as creating a domain-specific language, are non-trivial and often require repetitive tasks. This results in unnecessary increases of development time. The goal of this paper is to increase the productivity of modelers in their every day activities by automating the tasks they perform in current MDE tools. We propose an MDE-based solution where the user defines a reusable workflow that can be parametrized at run-time and executed. Our solution works for frameworks that support two level meta-modeling as well as deep metamodeling. We implemented our solution in the MDE tool AToMPM. We also performed an empirical evaluation of our approach and showed that we reduce both mechanical and thinking efforts of the user. The ideas and concepts of this paper were introduced at the MODELWARD conference [1] and are extended in this paper.

## 1 Introduction

Model-Driven Engineering (MDE) has been advocating faster software development times through the help of automation [2]. MDE technologies combine domain-specific languages (DSL), transformation engines and code generators to produce various software artifacts. Although some studies report success stories of MDE [3], some of the less satisfactory results include the presence of a plethora of MDE tools. Each tool defines its own development and usage process, which is a burden on the user who needs to adapt himself to every tool. To be successful, MDE needs tools that are not only well adapted to the tasks to perform, but also tools that increase the productivity of modelers in their day-to-day activities.

Modeling tools and frameworks, such as AToMPM [4], EMF [5], GME [6], and MetaEdit+ [7], provide many functionalities, such as DSL creation, model editing, or model transformations. Although based on common foundational principles, the process for performing these tasks differs greatly depending on the tool used. For example, to create a DSL in AToMPM [8], the language designer has to load the class diagram formalism and graphically build the metamodel. He generates the abstract syntax of the DSL from that metamodel by loading the compiler toolbar. Then he has to load the concrete syntax formalism and assign a concrete syntax to each individual class and association from the metamodel

by drawing shapes. He then generates the domain-specific modeling environment by loading the compiler toolbar. In contrast, the steps are different to create a DSL in EMFText [9]. The language designer first creates a new project by specifying the project settings in the wizard dialog. He then creates an Ecore diagram file and graphically builds the metamodel. He then needs to create a generator model from the metamodel file. To define the concrete syntax, he creates a file specifying the textual grammar. Once completed, he executes the generators to create the domain-specific environment that needs to be launched as a separate Eclipse instance initiated from the generated Java code.

Many of these activities involve repetitive tasks and a lot of user interactions with the user interface of the MDE tool. These are non-trivial activities. They involve long sequences of tasks, often repetitive tasks. Additionally, they require context-dependent decisions leading to a lot of user interactions with the user interface of the MDE tool. The processes to follow are complex for all users, whether they are language engineers (i.e., MDE savvy) or domain-specific modelers (i.e., end-users). They require heavy mental loads and tasks that are error-prone. In the end, users are spending more time on development than necessary. It is therefore mandatory to try to automate MDE tasks and processes as much as possible, thus decreasing the accidental complexity of the tools used and letting the user focus on the essential complexities of the domain problem.

To solve this issue, tools can implement automated workflows for each MDE activity that involves a complex process or repetitive tasks. Many of the tools already partially support this with the help of wizards [5] or scripts [10]. However, even these wizards become quite complex offering too many options that the user has to manually input each time he wants to repeat an activity, as in Eclipse based tools. There are also several languages to define processes, such as SPEM [11], but do not support their execution (or *enactment*) natively. Other executable process languages like BPEL [12] are too complex for the tasks we want to achieve in modeling tools. Workflow languages, such as UML activity diagrams, can be enacted [13], but the execution relies on programming individual actions which hampers porting a process from one tool to another.

We therefore propose to define a DSL, inspired from activity diagrams, that fits exactly the purpose of designing workflows for common tasks in MDE tools. The tasks encompass simple operations, such as opening, closing or saving models, and more complex tasks, such as generating the artifacts for a DSL. We noted that several tasks occur in different workflows, especially common operations e.g., open and close. Therefore we opted for a reuse mechanism, where the user defines workflows that can be parametrized at run-time to minimize the number of workflows to create. Since our solution follows the MDE paradigm, the execution of workflows is entirely modeled through model transformation. Ultimately, users spend less time performing the activity by focusing on essential model management tasks rather than wasting time interacting with the tool. The ideas and concepts of this paper were introduced at the MODELSWARD conference [1] and are extended in this paper.

The paper is organized as follows. In Sect. 2, we describe the details of our solution and discuss how we solved challenges we faced. In Sect. 3, we report

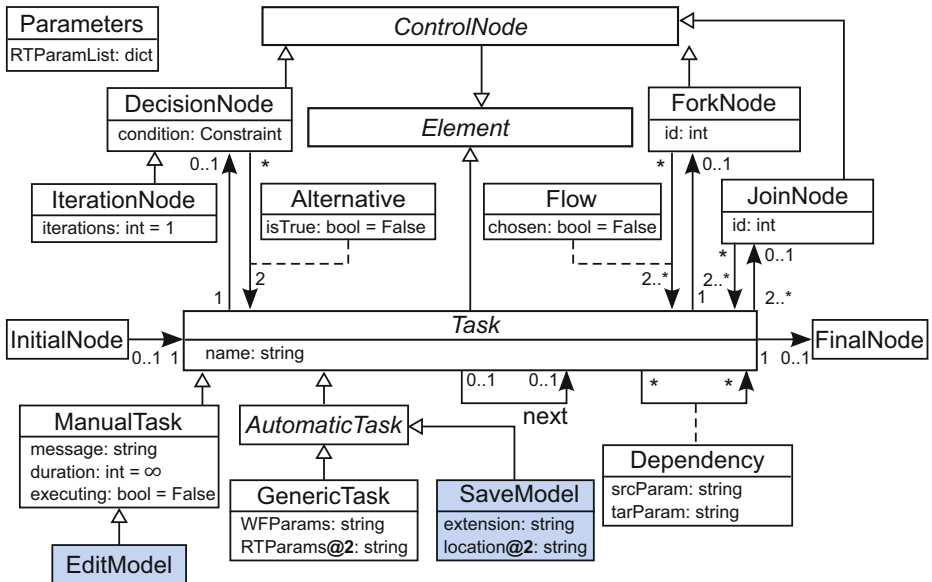
on the improved implementation of our approach in AToMPM. Specifically in Sect. 4, we discuss how model refactoring is automated. In Sect. 5, we perform a preliminary empirical evaluation of the impact our approach has on improving the user productivity in AToMPM. Finally, we discuss related work in Sect. 6 and conclude in Sect. 7.

## 2 Design of a Reusable Workflow Language

We propose an MDE-based solution where the user defines workflows that can be parametrized at run-time and executed. In this section, we describe a DSL that is adaptable to a specific modeling tool. We also describe the general process of how to design reusable workflows to semi-automate MDE activities. Furthermore, we discuss how to enact workflows using model transformation.

## 2.1 Language for Semi-automated Workflows

We model the DSL for defining activities that can be performed in MDE tools. A workflow is composed of tasks, to define concrete actions to be performed, and control nodes, to define the flow of tasks. The metamodel in Fig. 1 resembles that of a simplification of UML activity diagrams since, semantically, an instance of this metamodel is to be interpreted similarly to the control flow in UML activity diagrams. Additional well-formedness constraints are not depicted in the figure e.g., a cycle between tasks must involve an iteration node, there must be exactly one initial and one final node.



**Fig. 1.** Generic metamodel of workflows for modeling tools.

There are different kinds of tasks in an MDE tool. As for any modern software, there are tasks specific to the user interface, such as opening, closing, and saving models or windows. There are also tasks that are specific to models, such as editing (CRUD operations) models, constraints, or transformations. There are also tasks that are specific to the particular modeling tool used, such as loading or executing a transformation, generating code from a model, or synthesizing a domain-specific environment from a DSL. Furthermore, we want to automate users' activities as much as possible, therefore most of the tasks are automatic: they do not require human interaction. For example, loading a formalism to create a metamodel is (e.g., `Ecore` in EMF or `Class Diagrams` in AToMPM) is a task that can be automated, since the location of that formalism is known. Shaded classes in Fig. 1 (`SaveModel` and `EditModel`) are examples of tasks that may vary from one MDE tool to another. Otherwise, this is a generic metamodel implementable in any MDE tool.

Nevertheless, some tasks are hard, even impossible, to automate and thus must remain manual. These are typically tasks specific to a particular model, such as deciding what new element to add in the model. A message is specified to guide the user during manual tasks. A maximum duration can also be specified to limit the time spent on a manual task.

A workflow conforming to this metamodel starts from the initial node and terminates at the final node. Tasks can be sequenced one after the other. A decision node can be placed to provide alternative flows (one true and one false) depending on a Boolean condition evaluated at run-time. Repetitions are possible with an iteration node. This node repeats the flow along the true alternative as long as the condition is satisfied. A common condition is to limit the number of iterations: e.g., `self.iterations <= 2`. The cycle ends when either the specified number of iterations is reached or a terminating condition is satisfied. Fork and join nodes provide non-determinism when the order of execution of tasks is not relevant. Fork node is a control node that splits a flow into multiple concurrent flows and join node is a control node that synchronizes multiple flows. These correspond to the common basic control flow patterns for workflows [14]. Although not supported in our current implementation, tasks may be executed concurrently, except if the concurrent tasks are manual.

## 2.2 Parameters

One issue that may slow down the development time of users using workflows, is that many tasks require parameters. For example, the task `SaveModel` requires the location of where to save the model (path and name) and the extension to be used. The extension is generally known from the context of the workflow. For example, a generic model ends with `.ecore` in EMF and `.model` in AToMPM, but a domain-specific model may have a specific extension in EMF. The designer of the workflow can thus set the value of this attribute at design-time. However, the location of the model is generally unknown to the workflow designer because it is a decision often left at the discretion of the domain user. We therefore

distinguish between workflow parameters that are fixed for all executions of the workflows and run-time parameters that are specific to individual executions of the workflow.

Within the same workflow, several tasks may share the same parameters. Workflow parameters are specified once per workflow. However, run-time parameters must be manually specified each time the workflow is executed. Therefore, a **Dependency** link can be specified between different tasks that share the same run-time parameters. A dependency link specifies which attribute from the target task gets its value from an attribute in the source task. For example, the location of the **SaveModel** task is the same as the location of the **OpenModel** when saving a model we just opened and modified.

### 2.3 Activities as Workflows

To set the values of run-time parameters, we need an intermediate model of workflows that is an instance of the metamodel presented, but where some parameters are left for further assignment. As explained in [15], the commonly used technique of two-level metamodeling does not allow us to represent this need.

An attractive solution is to apply techniques from deep metamodeling [16], and in particular, the approach defining metamodels with potency [17]. We assign a potency of 2 to attributes representing run-time parameters and a potency of 1 to those representing workflow parameters, as depicted in Fig. 1. This way, the workflow designer only needs to create one workflow for saving models with the extension set to e.g., `.model` and the user can execute the workflow only caring of the location where to save the model and not bother what the right extension is. In this setup, an instance of the workflow metamodel in Fig. 1 is a workflow. A workflow is itself the metamodel of its instantiation at run-time. The *enactment* of a workflow therefore consists in providing the run-time parameters to a workflow and executing it. These definitions are consistent with what the Workflow Management Coalition specifies [18].

### 2.4 Workflow Enactment by Model Transformation

In this section, we describe how workflows are instantiated with run-time parameters and executed.

**Deep Instantiation.** The issue with the above solution is that not many modeling frameworks (e.g., AToMPM<sup>1</sup> and EMF) support deep metamodeling with potency like metadepth [20] or Melanee [21] do. Therefore, we propose a workaround to enact workflows by emulating deep metamodeling with potency for tools that do not natively support it. The solution is to add a **Parameters** class to the metamodel that is instantiated once per workflow enactment. Its

<sup>1</sup> In [19], the authors proposed a deep metamodeling solution for the Modelverse of AToMPM, but no usable implementation was available at the time of writing this paper.

attributes are populated dynamically for the enactment. They consist of all the run-time parameters of every task in the workflow. The parameter object is used to generate a wizard prompting for all run-time parameters needed in the tasks of a workflow.

Once a workflow has been created by the workflow designer, a user can enact the workflow. He creates a parameter object to specify run-time parameters and executes the workflow. We have modeled the enactment of workflows by model transformation. Figure 2 depicts the transformation in MoTif [22], a rule-based graph transformation language in AToMPM. Rules are defined with a pre-condition pattern on the left and a post-condition pattern on the right. Constraints **Const** and actions **Act** on attributes are specified in Python. A scheduling structure controls the order of execution of rules. Figure 2 shows the two-step transformation that retrieves all run-time parameters of the workflow. The transformation on the left of the figure populates all attribute fields of the parameter object (the icon with two gears) by visiting each task in the workflow model. The first rule makes sure a depended run-time parameter is not added to the parameter list of the parameter object. For each parameter, we store the task type, its task name (in case multiple instances of the same task type are in the workflow), and the name of the parameter. We make use of the **setAttr** and **getAttr** functions that allows us to get and set attribute values using the attribute name as a string. This information is then used to render a wizard prompting for their corresponding values to the user. Once the user enters all parameters, the transformation on the right of the figure copies the values entered in the source run-time parameters to the target run-time parameters. This makes sure that all run-time parameters of all tasks are set. Note that the transformations uses FRules to make sure that each task is visited exactly once, which is why no negative application condition is needed.

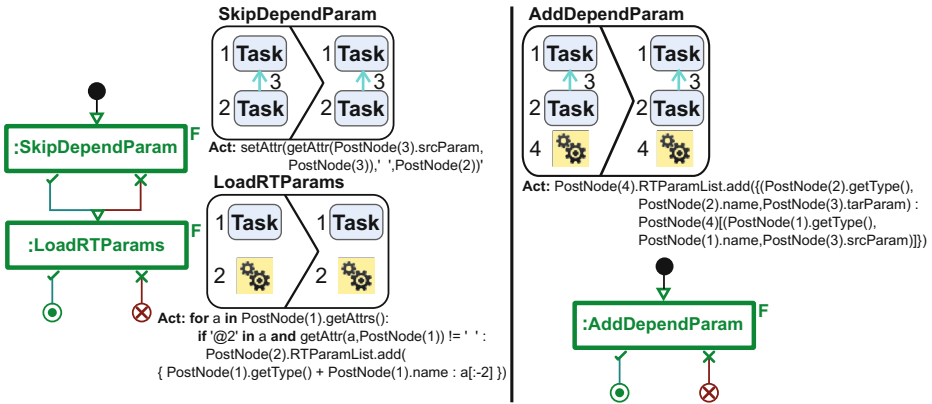
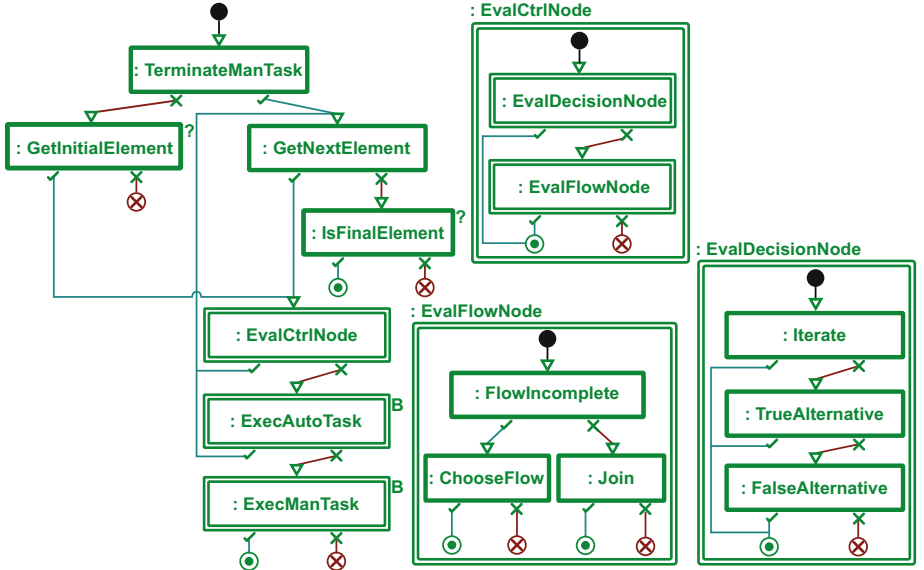


Fig. 2. Transformation for loading run-time parameters in MoTif.

**Execution.** With all run-time parameters set, there are two ways to execute the workflow. One is to transform the workflow into a model transformation that gets executed, as done in [23]. In this case, a higher-order transformation takes as input the workflow and parameter object, generates a rule for each task, and schedules the rules according to the order of the tasks in the workflow. This is possible in MoTif since rules and scheduling are specified in separate models. Although this approach has the advantage to reuse built-in execution mechanisms from the MDE tool, a new transformation must be generated for each workflow and, in particular, if the designer makes changes to the workflow model.

In this work, we have implemented an alternative solution: we define the operational semantics of a workflow and execute it as a simulation. Figure 3 illustrates the overall structure of this transformation and Fig. 4 depicts some of the rules. The process starts from the element (task or control node) marked with the initial node. The rule **GetInitialElement** is responsible for this and specifies only a pre-condition. The general idea is that then, each task to process each element in the order of the workflow by advancing the **current** pointer called pivot in MoTif, with the rule **GetNextElement**. The simulation ends when the final node is reached, satisfying the rule **IsFinalElement**. Executing an automatic task, such as save model depicted in rule **ExecuteSaveModel**, is performed by calling the corresponding API operation of the MDE tool with the corresponding run-time parameters. We assume that the MDE tool offers an API for interact-



**Fig. 3.** Control structure of the transformation in MoTif that executes a workflow.

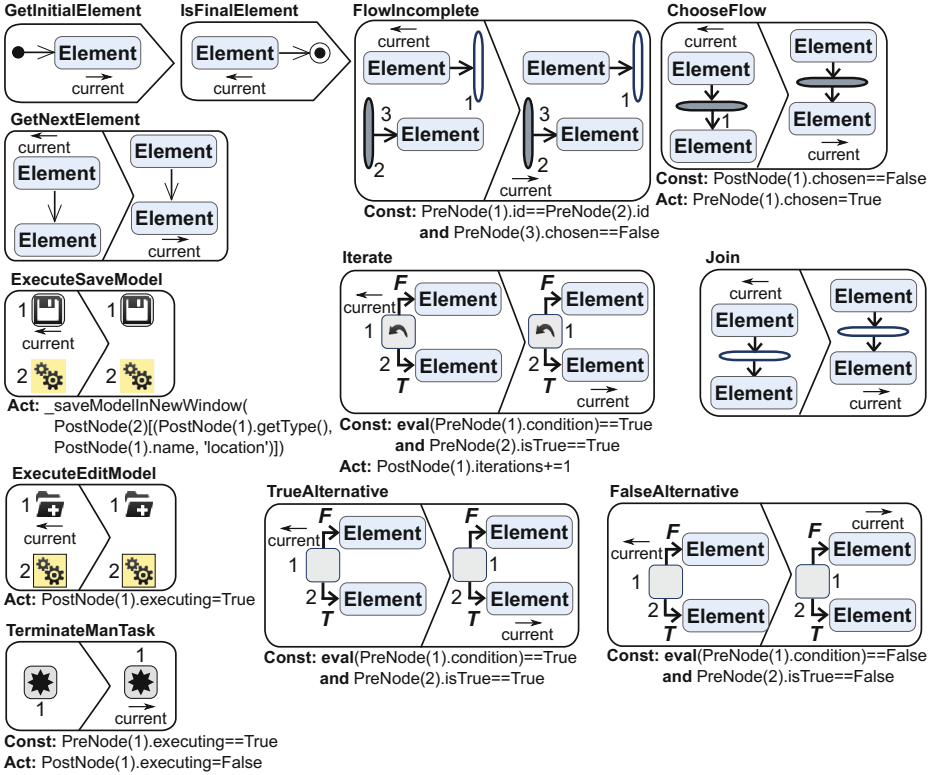


Fig. 4. Transformation rules in MoTif that execute a workflow.

ing with it programmatically (e.g., Python API for AToMPM and Java API for EMF).

When a control node is the current element to process, we need to decide on which element is next to be processed. For a decision node, if the condition is true, then the next element along the true branch is selected. Otherwise, it is the next element along the false branch. This assignment is the same for iteration nodes, except that the `iterations` count is incremented as long as the condition is satisfied. In our implementation, the semantics of a fork is to choose non-deterministically one of the flows, execute all tasks in that flow in order, and then choose another flow. The rules in `EvaluateFlowNode` ensure this logic: when a join node is reached, we make sure that all flows outgoing from the corresponding fork are complete as expressed by rule `FlowIncomplete`.

This process runs autonomously as long as there are automatic tasks. However, manual tasks require interruption of the transformation in real-time so that the user can complete the task at hand and then resume the transformation. Automating such a process requires to be able to pause and resume the transformation from the rules being executed. Although some transformation languages



support real-time interruption [24], most do not. Therefore, as depicted in Fig. 3, we extend the logic to handle manual tasks separately. If the next task to execute is manual, the corresponding rule simply flags the task as executing, as rule **ExecuteEditModel** shows, and the transformation terminates. The user notifies the MDE tool that his manual task is complete by restarting the transformation. Consequently, the transformation executes the first rule **TerminateManTask** which resumes the execution from the task that was last marked as executing. The **executing** attribute for manual tasks allows the workflow model to keep track of the last manual task executed after the transformation is stopped.

## 2.5 Extensions and Exceptions

The approach presented here is evolution safe. MDE tools evolve with new features added. If a new feature is available via the API and is needed in an workflow, then there are only two steps the designer is required to perform to support that feature. He shall add a new sub-class of automatic or manual task in the metamodel of Fig. 1 and add a rule under **ExecAutoTask** or **ExecManTask** in Fig. 3 that calls the appropriate API function to perform the operation. **ExecAutoTask** (respectively **ExecManTask**) is a BRule that contains all the rules to execute automatic (respectively manual) tasks. BRules execute at most one of their inner rules unless none of them are applicable. The modularity of this design reduces significantly the effort of workflow designers who wish to provide additional tasks available via new features of the MDE tool.

Although it is common to explicitly model exceptional cases in workflows [25, 26], we have decided not to do that at the workflow model level. Exceptions can only occur if a task execution fails because the user is constrained to do exactly what the workflow allows as next action. In this version of our implementation, if an exception occurs, the workflow execution stops at the failing task in the workflow, as depicted by the circled crosses in Fig. 3. The user must then manually recover from the error and restart the execution of the workflow. Nevertheless, run-time parameters are retained.

## 3 Implementation in AToMPM

We implemented a prototype in the MDE tool AToMPM [4], since it offers a graphical concrete syntax for DSLs, which is best suited for workflow languages, and a backdoor API to programmatically interact with the tool in headless mode. Nevertheless, our approach can be implemented in any MDE tool as long as it offers an accessible API to perform operations that their user interface allows to. We implemented the workflow DSL following the metamodel in Fig. 1. Figure 5 shows the graphical representation used for each task, each control node, and parameter object.

We analyzed several processes and noted the user interactions needed to perform each task, e.g., creation of DSL. We had to decide on what level of granularity we want to present tasks. One option is to go to the level of mouse movements

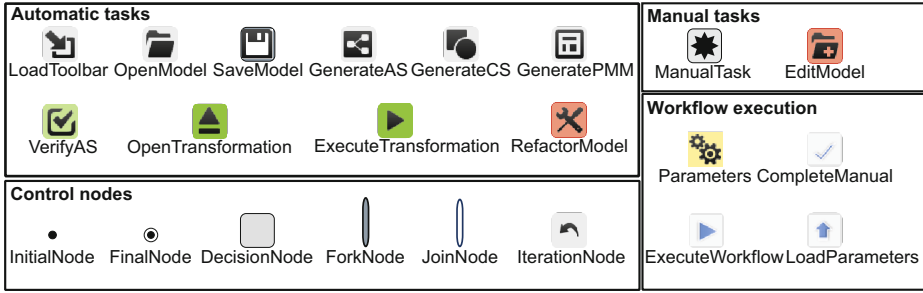


Fig. 5. Concrete syntax of the workflow DSL in AToMPM.

(graphically moving objects), clicks (selections), and keystrokes (textual editing). Although this would enable us to model nearly any user interaction AToMPM allows for, this would make the workflows very verbose and complex for designers. We therefore opted for tasks to represent core functionalities instead. Subsequently, the most common tasks we noted are opening models, loading toolbars and formalisms, saving models, generating concrete and abstract syntax of DSLs, as listed in Fig. 5. All these operations can be automated, since they require a location as run-time parameter. **SaveModel** also has a workflow parameter for the extension of the model file. Additionally, a task to edit models is needed, but cannot be automated since it is up to the user to create or edit the model.

### 3.1 Process

Our prototype is to be used as follows. The designer defines workflows by creating instances of the workflow DSL. A user (a language engineer in this example) then selects which workflow he desires to enact. To set the run-time parameters, he pushes the **LoadParameters** button. This creates an instance of the parameter object and pops up a dialog prompting for all required parameters, following the transformation from Fig. 2. Upon pushing **ExecuteWorkflow** button, the simulation (presented in Fig. 3) executes the workflow autonomously. When a manual task is reached, a new AToMPM window is opened with all necessary toolbars pre-loaded. A message describing the manual task to perform is displayed to the user and the simulation stops. After the user completes the task, he pushes the **CompleteManual** button. Then, the window closes and the simulation restarts.

### 3.2 Example Workflow for Creating a DSL

Figure 6 shows the workflow that specifies how to create a DSL and generate a modeling environment for it in AToMPM. The first task is **LoadToolBar**. Its location parameter is already predefined with the class diagram toolbar, since this is the standard formalism with which one creates a metamodel in AToMPM. The following task is **EditModel**. In this manual task, the user creates the metamodel of the DSL using class diagrams. Once this is complete, the workflow restarts

executing from that task and proceeds with **SaveModel1**. This task requires a run-time parameter to specify the location of where the metamodel is saved. The user sets the value in the popup dialog wizard. Now that the metamodel is created, a fork node proposes two flows: one for creating the concrete syntax of the DSL and one to generate the abstract syntax from the metamodel. Recall that the simulation chooses one flow and then the other in no specific order. Suppose the former flow is chosen. Then, a **LoadToolBar** task is executed to load the concrete syntax toolbar, the standard formalism in AToMPM. This is followed by an **EditModel** so the user can manually create the shapes of each element of the metamodel. Once this is complete, the workflow restarts and proceeds with a **SaveModel** task. Recall that the location is a run-time parameter to save the concrete syntax model with a predefined extension. In the popup dialog, we distinguish between different task with their type, and in this case their name (1 and 2). The following task in this flow is **GenerateCS**. It takes as run-time parameter the location of where the generated artifact must be output. Specifically, the name used will be also the name of the toolbar that will be used to create a model with this DSL. Therefore, the location of the generated concrete syntax is the same as the location of the concrete syntax model the user created manually. The dependency link prevents the user from having to duplicate parameter values in the wizard. When the join node is reached, the simulation notices that the second flow was not executed yet. Therefore the next task to be executed is **GenerateAS**. Its location parameter uses the same value of the location attribute of **SaveModel 1**, as depicted by the dependency link between these two tasks. When the join node is reached again, this time all flows were executed and proceeds with the final task **LoadToolBar 3**. As stated before, its location parameter use the same value of the location attribute of **SaveModel 2**. The simulation ends on a new window open with the new DSL loaded, ready for the user to create his domain-specific model.

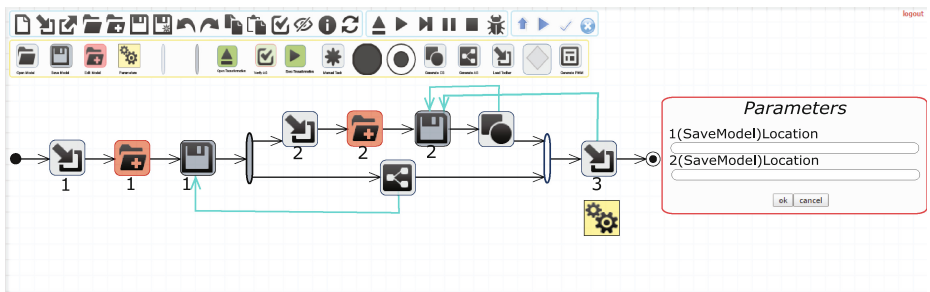


Fig. 6. Workflow to create a DSL.

## 4 Automating Refactoring Tasks

Refactoring is common operation on modeling artifacts that improves the structure of a model while preserving its external behavior [27]. In MDE, refactoring is either done manually on a model or through the application of a model transformation [28]. There exists several techniques to perform refactoring on generic or domain-specific models [29], and even a catalog of refactoring patterns on metamodels [30].

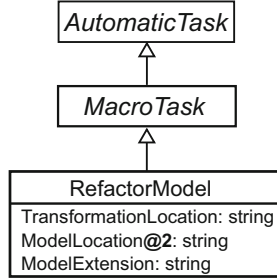


Fig. 7. Generic metamodel of Refactoring Model.

Refactoring is an activity that can be automated in our workflow system. By default, this can be done through a manual task. However, we also support automating this task for the user. To do so, we extend the metamodel of Fig. 1 with the concept of a **MacroTask** as depicted in Fig. 7. A macro task is an implicit workflow of other tasks. For example, as illustrated in Fig. 8, **RefactorModel** is decomposed into opening the model to refactor, loading the transformation that implements the refactoring, and executing that transformation on the model. For the **RefactorModel** task, the location of the transformation is a workflow parameter specified by the workflow designer. Additionally, this task requires the location of the model to refactor, but this is a run-time parameter that the user specifies. The extension of the model is generally known from the context of the workflow.

A macro task serves as syntactic sugar to simplify the workflow of the user. The semantics of a macro task is modeled by a transformation executed during the simulation in Fig. 3. The implicit transformation that is executed for

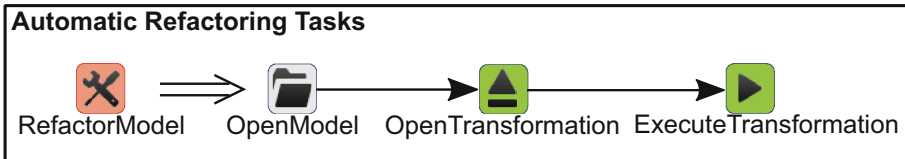


Fig. 8. Generic metamodel of Refactoring Model.

**RefactorModel** can be defined on the meta-metamodel level (e.g., class diagram in AToMPM or Ecore in EMF) so that it is syntactically applicable on any given model. The burden is on the user who needs to define a meaningful transformation that can be applied on the desired model. For example, if the model is a metamodel, then a refactoring can add a unicity constraint. If the model is a concrete syntax assignment, then a refactoring can create a default concrete syntax to every class of the metamodel.

## 5 Evaluation of the Improvement of MDE Activities

### 5.1 Research Question

The goal of the experiment is to determine whether the productivity of the user is increased when performing complex or repetitive tasks. Thus, our research question is “is the time for mechanical and cognitive efforts of the user reduced when automating activities with workflows?” Therefore, we conduct the experiment to verify that these efforts are reduced when using our approach versus when not.

### 5.2 Metrics

The total time  $T$  spent by a user to perform one activity is one way to quantify the effort the user produces.  $T$  is mainly made up of the mechanical time  $T_m$  (hand movements) and cognitive effort time  $T_t$  (thinking time) of the user, thus  $T = T_m + T_t$ , assuming there are no interruptions or distractions.

Since AToMPM only presents a web-based graphical user interface and most interactions are performed with a mouse, we can apply Fitts Law [31] to measure the time of mouse movements  $t_{FL} = a + b \times \log_2(1 + D/S)$ .  $D$  is the distance from a given cursor position to the position of a widget to reach (e.g., button, text field) and  $S$  is the smallest value of the width or height of the widget. We denote  $T_{FL}$  as the sum of all the  $t_{FL}$  for each useful mouse movement to perform one activity.

Another useful metric we noted for the mechanical effort is the number of clicks  $c$  needed to complete the activity. Relying on empirical data from an online benchmark [32], the average time to click reactively is 258 ms. Thus we denote  $T_c = 258 \times c$  the time spent clicking during an activity.

Therefore a rough estimate of the time spent on mouse actions in an activity is  $T_m = T_{FL} + T_c$  for every straight line distance  $D$  between two clicks and the size  $S$  of the widget at every even click.

Delays between mechanical actions is a rough estimate of the time the user spent thinking during the activity. Hence, we deduce the thinking time  $T_t = T - T_m$ .

Finally, we measure the complexity  $N$  of a task by the number of automatic tasks it requires the user to perform.

These metrics are far from accurate, but serve at least as a preliminary evaluation of our approach to discard the null hypothesis:  $T_m$ ,  $T_c$  and  $T_t$  are smaller for performing an MDE activity in AToMPM using workflows than without workflows.

### 5.3 Experimental Setup

We performed all experiments on a 15.6" laptop monitor with a resolution of  $1920 \times 1080$ . The machine was an ArchLinux virtual machine using 2 cores and 4GB of RAM, running on Windows 10 quad-core computer at 2.4GHz with 16GB of RAM. Given this performance, we neglected the computation time of AToMPM triggered by each click. To keep a fair comparison, the experiments using the workflow did not take into account the mouse activity and time spent during manual tasks. This is the time after the simulation terminates and before the notification from the `CompleteManual` button is received.

### 5.4 Data Collection

To calculate  $t$  using Fitts law, the coefficients  $a$  and  $b$  must be determined empirically. For that, we recorded the straight line distances between meaningful clicks (e.g., center of canvas to toolbar button) as well as different sizes of clickable elements (e.g., model elements on the canvas) in AToMPM. We recorded 12 distances ranging from 79 to 1027 pixels and 5 sizes ranging from 20 to 305 pixels. We then placed on an empty screen a point and a rectangle of sizes and at distances that correspond to these measurements. We measured the time it took to click on the initial point and move the cursor as fast as possible to click inside the opposite rectangle. This data collection was performed by the first author who is an expert in AToMPM. We repeated each of the 57 cases 20 times (excluding those where  $D \leq S$ ). The maximum variation in the same case was less than 9%. We determined by regression analysis the values  $a = 166.75$  and  $b = 155.93$  with correlation  $R^2 = .9106$  with a median and average margin of error of 8%.

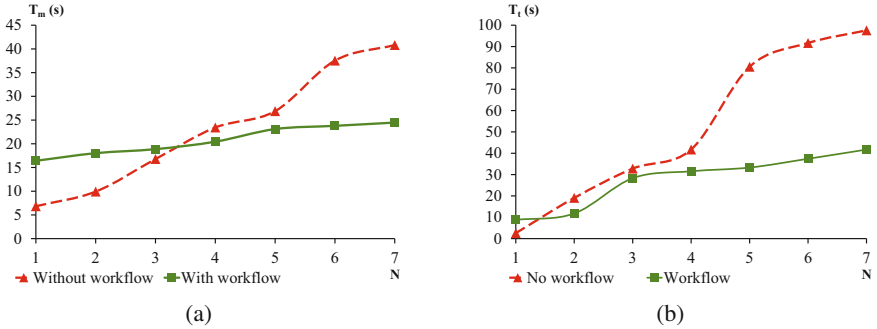
In our prototype, we implemented the five most common tasks in AToMPM shown in Fig. 5. There is an infinite number of possible combinations of these tasks because tasks can be repeated and the order matters. Therefore, we reduced the number of cases to only meaningful combinations of tasks in AToMPM. We identified 4 meaningful for activities with one task (compiling the concrete syntax requires a model to be opened), 9 for activities with two tasks (e.g., open then save model), 13 for activities with three tasks, 4 for activities with four tasks, 5 for activities with five tasks, 3 for activities with six tasks, and 3 for activities with seven tasks. Hence we ran our experiments on 38 distinct activities varying up to seven automatic tasks.

The most complex activity we evaluated is for the creation of a DSL in AToMPM modeled with the workflow in Fig. 6, consisting of seven automatic tasks. The workflow starts by loading the `Class Diagram` formalism. It lets the user manually create the appropriate class diagram model to define the

metamodel. When the user completes that task, the metamodel is saved (location provided at run-time) and the abstract syntax is generated. Then the **ConcreteSyntax** formalism is loaded and the user creates the shapes for links and icons. When the user completes that task, the concrete syntax model is saved (name provided at run-time) and the **GenerateCS** task generates the code for the new DSL environment. Finally, the new formalism is loaded in a new window showing the new generated DSL environment to the user. Note that in this situation, the first **LoadToolBar** object does not require a run-time parameter, but a workflow parameter for the location of the **Class Diagram** formalism. We therefore suggest to create two classes in the metamodel for the same task when we want to give the option to set either run-time or workflow parameters depending on the context.

## 5.5 Results

The two plots in Fig. 9 report the time performances for each case. We aggregated the times by the number of tasks because there was very few variability between activities with the same number of tasks: the highest coefficient of variability 20% was obtained for activities with three tasks since this was the most populous set, while all the others remained under 5%. Both plots confirm that the use of workflows does reduce the time to perform the activity, as the complexity of the activity increases.



**Fig. 9.** Mechanical (a) and cognitive (b) efforts with respect to the number of tasks in a workflow.

The results obtained correspond to what one would expect when adding automation in a development process. The mechanical effort is greater when using workflows for simple activities that have up to three tasks. However, after that point, the mechanical effort remains almost identical as the number of tasks increases. This behavior, depicted in Fig. 9(a), is due to the overhead to open the appropriate workflow and set all run-time parameters. The reason why  $T_m$  plateaus after  $N = 5$  is that the only mechanical effort needed is to specify

additional run-time parameters. However, this is done by typing the values with the keyboard which we haven't taken into account in this experiment. When performing the experiments, we noted that the slowest task performed manually was for loading toolbars.

Figure 9(b) reports on the non-mechanical effort needed by the user to perform each activity. We note a trend similar to the mechanical effort. However, the flip point where less effort is needed when using workflows occurs as early as activities with more than one task. The cognitive effort increases linearly for activities with more than three tasks. An interesting result is that, when not using workflows, the cognitive effort is always greater than the mechanical effort for  $N > 1$  and that gap keeps on increasing as there are more tasks. On the contrary, when using workflows, the mechanical effort is greater for activities with up to two tasks, but when the cognitive effort is greater for  $N > 2$ , the gap remains almost identical. When performing the experiments, we noted that most of the time was spent searching on the screen to select toolbars to load, even for an expert user who knows exactly their locations.

To complement this information, Table 1 details each metric for the most complex activities we evaluated. It shows that, although using workflows improves all the metrics, the cognitive time is the most improved component.

**Table 1.** Time measurements in seconds and improvements when using workflows for  $N = 7$  tasks.

	$T$	$T_{FL}$	$T_c$	$T_m$	$T_t$
No workflow	138	29	11	41	98
Workflow	66	18	6	24	42
Improvement	52%	38%	45%	41%	57%

We conclude that our hypothesis is verified and answer our research question: for the extent of the experiments we conducted, the time for mechanical and cognitive efforts of the user is reduced when automating activities with our approach by half.

## 5.6 Threats to Validity

There are several threats to the construct validity of this preliminary evaluation. First, the metrics we used are not sufficient to assess the complete mechanical effort. Keystrokes can also be taken into account since there is an effort needed to set the values of run-time parameters. However, the length of the string of each depends on the file paths of the host machines and the operating system used. We discarded this metric for its lack of generalization. Further mechanical metrics could be used such as eye movements, but we lacked the proper hardware to perform eye-tracking experiments. We further mitigated these threats by using Fitts Law to achieve an objective measure of time mouse movements.



We measured cognitive effort by considering it as all non-mechanical effort, which is not a completely true statement. Otherwise, this would have required more fine grained measurements of brain activity. We also did not include the time and effort for manual tasks, which may have a negative influence on the results if they take longer than the automatic tasks. The data collection was performed by only one person, but this was only necessary to calculate  $t$  since all other metrics are obtained using Fitts Law, without needing to perform the activities. This threat only affects the absolute time, but does not affect the improvement ratio.

With respect to threats internal validity, the selection and configuration of the tools for time measurements has a weak influence on the results. We calibrated the parameters based on a pilot experiment and our experience. However, this should not strongly affect the time because we took care of configuring the tools in a way that corresponds to the empirical data from an online benchmark. We also pre-processed inconsistent times (e.g., clicks outside target) in order to eliminate false positives. Nevertheless, this only reduces the chances that we can answer our research question positively.

As far as threats to external validity are concerned, the activities were obviously not sampled randomly from all possible MDE tools activities, but we relied on our knowledge in MDE tools. Hence, the set of activities is not completely representative. The results of this study can only be generalized to the extent of AToMPM. Nevertheless, all five tasks we considered are part of the most common activities in the majority of MDE tools, such as EMF. We further mitigated this threat by including tasks with different complexity (i.e., Open Model vs Compile Abstract Syntax) and focusing on their meaningful combinations.

## 6 Related Work

A lot of work can be found in the literature on workflow definition and enactment [33–35]. In [36], the authors proposed a textual DSL for workflow definition that supports sequencing and iteration. It is not meant to be enacted, but serves as specification for subsequent code generators. Workflow enactment has been particularly applied in process modeling.

Various techniques exist to service the execution of workflows, such as distributing the execution on the cloud [37,38]. However, none of these approaches models workflow enactment explicitly as we did using model transformation.

We proposed a model transformation as a novel workaround for tools that do not support deep instantiation of metamodels. An alternative is to define metamodels following the Type-Object pattern [39] where both types and instances are explicitly modeled in the metamodel. This is similar to the notion of clabject [40] which generalizes this approach.

From an implementation point of view, the closest work to ours automates transformation chains in AToMPM [23]. They developed a formalism transformation graph (FTG) that specifies a megamodel indicating the transformations between languages and a process model (PM) that specifies the control and data

flow to schedule the order of execution of model transformations. The execution of an FTG+PM instance is modeled as a higher-order transformation that converts the FTG+PM model into a model transformation instance, whereas our approach executes workflows by simulation. The authors also distinguish automatic actions from manual ones, but the latter are not modeled in the transformation.

Similarly to FTG+PM, Wires [41] supports the specification and execution of model transformation workflows. Wires is graphical executable language for ATL transformations that provides mechanisms to create model transformations chains. Kepler [42] is a tool to create and execute scientific workflows. Since it is based on the Ptolemy II multi-paradigm simulation system, a coordinator must be hand-written in Java to define the semantics of the workflow, unlike our approach that makes use of model transformation.

In our approach, activities essentially encapsulate model management tasks. The Epsilon language suite [43] can be used to perform model management tasks such as CRUD operations, transformations, comparisons, merging, validation, refactoring, evolution, and code generation. To combine and integrate these different tasks into workflows, the user defines Ant scripts. In our approach, users define workflows in a DSL specific to the features the MDE tool provides. As such, it reduces accidental complexity imposed by Ant and is accessible to a broader set of users that do not know Ant. One particular language is the Epsilon Wizard Language (EWL) [44] whose purpose is to refactor, refine, and update models. EWL allows users to define wizards that serve as encapsulation of EOL scripts, the action language in Epsilon. Wizards are similar to activities in our case. EWL provide feedback that can drive the execution of a model management operation using a context-independent user input. It is a command line user input interface. In our approach, the user-input method is a popup dialog with several parameters. Their approach has a more fine-grained wizard selection process, since a wizard can have a guard that must be satisfied in order to execute it. Nevertheless, EWL does not support the explicit modeling of manual tasks. EWL is especially designed for refactoring models automatically. These model refactorings are applied on model elements that are explicitly selected by the user. Typical supported refactoring patterns include adding the stereotypes, attributes and operations. EWL has constructs specifically to refactor model elements. In our approach, workflows rely on a model transformation to express the modification to the model. Therefore the user only needs to specify the model, and not individual model elements.

## 7 Conclusion

In this paper, we presented a model-based environment for automating daily activities of language engineers and domain-specific modelers. Designers define workflow templates conforming to a DSL to increase the productivity of users. Users enact workflows to perform tasks automatically. Our framework also supports the integration of manual tasks. The execution of workflows is entirely

modeled as a model transformation, making it reusable and portable on various MDE tools. Preliminary results of our prototype indicate that, using workflows, users reduce cognitive and mechanical effort to perform common activities in the MDE tool AToMPM.

We are integrating more features of AToMPM in our prototype to allow designers define workflows for nearly any interaction process the tool can do. As future work, we plan to implement this approach in other MDE frameworks, such as EMF, in order to further generalize the reusability aspect of the metamodel of activities and their enactment by model transformation.

## References

1. Gamboa, M.A., Syriani, E.: Automating activities in MDE tools. In: Model-Driven Engineering and Software Development, SciTePress, pp. 123–133 (2016)
2. Schmidt, D.C.: Model-driven engineering. *IEEE Comput.* **39**, 25–31 (2006)
3. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. *IEEE Softw.* **31**, 79–85 (2014)
4. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., Ergin, H.: AToMPM: a web-based modeling environment. In: Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition, MODELS 2013, vol. 1115, pp. 21–25. CEUR-WS.org (2013)
5. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison Wesley Professional, Boston (2008)
6. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing, WISP 2001, vol. 17 (2001)
7. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+ a fully configurable multi-user and multi-tool CASE and CAME environment. In: Constantopoulos, P., Mylopoulos, J., Vassiliou, Y. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 1–21. Springer, Heidelberg (1996). doi:[10.1007/3-540-61292-0\\_1](https://doi.org/10.1007/3-540-61292-0_1)
8. AToMPM tutorial (2013). <http://www.slideshare.net/eugenesyriani/atompm-introductory-tutorial>. Accessed 07 Aug 2015
9. EMFText screencast (2014). [http://www.emftext.org/index.php/EMFText\\_Getting\\_Started\\_Screencast](http://www.emftext.org/index.php/EMFText_Getting_Started_Screencast). Accessed 07 Aug 2015
10. JetBrains MPS (2015). <https://www.jetbrains.com/mps/> Accessed 07 Aug 2015
11. OMG: Software & Systems Process Engineering Metamodel specification 2.0 edn. (2008)
12. OASIS: Web Services Business Process Execution Language, 2nd edn. (2007)
13. Syriani, E., Ergin, H.: Operational semantics of UML activity diagram: an application in project management. In: RE 2012 Workshops, pp. 1–8. IEEE (2012)
14. Russell, N., van der Aalst, W., ter Hofstede, A., Mulyar, N.: Workflow Control-Flow Patterns: A Revised View. Technical report BPM-06-22, BPM Center (2006)
15. Gonzalez Perez, C., Henderson Sellers, B.: Metamodelling for Software Engineering. Wiley Publishing, Hoboken (2008)
16. Lara, J.D., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* **24**, 1–46 (2014)
17. Atkinson, C., Kühne, T.: The essence of multilevel metamodeling. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 19–33. Springer, Heidelberg (2001). doi:[10.1007/3-540-45441-1\\_3](https://doi.org/10.1007/3-540-45441-1_3)

18. WMC: Terminology and glossary. Technical report, WFMC-TC-1011, Workflow Management Coalition (1999)
19. Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., Kühne, T.: Multi-level modelling in the modelverse. In: Workshop on Multi-Level Modelling, MULTI 2014, vol. 1286, pp. 83–92. CEUR-WS.org (2014)
20. Lara, J., Guerra, E.: Deep meta-modelling with METADEPTH. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13953-6\\_1](https://doi.org/10.1007/978-3-642-13953-6_1)
21. Atkinson, C., Gerbig, R.: Melanie: multi-level modeling and ontology engineering environment. In: International Master Class on Model-Driven Engineering: Modeling Wizards, MW 2012, pp. 7:1–7:2. ACM (2012)
22. Syriani, E., Vangheluwe, H.: A modular timed model transformation language. J. Softw. Syst. Model. **12**, 387–414 (2011)
23. Lúcio, L., Mustafiz, S., Denil, J., Vangheluwe, H., Jukss, M.: FTG+PM: an integrated framework for investigating model transformation chains. In: Khendek, F., Toeroe, M., Gherbi, A., Reed, R. (eds.) SDL 2013. LNCS, vol. 7916, pp. 182–202. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-38911-5\\_11](https://doi.org/10.1007/978-3-642-38911-5_11)
24. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with time for simulation-based design. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 91–106. Springer, Heidelberg (2008). doi:[10.1007/978-3-540-69927-9\\_7](https://doi.org/10.1007/978-3-540-69927-9_7)
25. Russell, N., Aalst, W., Hofstede, A.: Workflow exception patterns. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 288–302. Springer, Heidelberg (2006). doi:[10.1007/11767138\\_20](https://doi.org/10.1007/11767138_20)
26. Syriani, E., Kienzle, J., Vangheluwe, H.: Exceptional transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 199–214. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13688-7\\_14](https://doi.org/10.1007/978-3-642-13688-7_14)
27. von Pilgrim, J., Ulke, B., Thies, A., Steimann, F.: Model/code co-refactoring: an MDE approach. In: Automated Software Engineering, pp. 682–687. IEEE (2013)
28. Mens, T.: On the use of graph transformations for model refactoring. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 219–257. Springer, Heidelberg (2006). doi:[10.1007/11877028\\_7](https://doi.org/10.1007/11877028_7)
29. Zhang, J., Lin, Y., Gray, J.: Generic and domain-specific model refactoring using a model transformation engine. In: Beydeda, S., Book, M., Gruhn, V. (eds.) Model-Driven Software Development, pp. 199–217. Springer, Heidelberg (2005)
30. Metamodel refactoring catalog (2016). [http://www.metamodelrefactoring.org/?page\\_id=584](http://www.metamodelrefactoring.org/?page_id=584). Accessed 19 May 2016
31. MacKenzie, I.S.: Fitts’ law as a research and design tool in human-computer interaction. Hum.-Comput. Interact. **7**, 91–139 (1992)
32. Benchmark, H.: (2015). <http://www.humanbenchmark.com/tests/reactiontime/statistics>
33. WMC: Process Definition Interface - XML Process Definition Language 2.00. Technical report, WFMC-TC-1025, Workflow Management Coalition (2005)
34. Mahmud, M., Abdullah, S., Hosain, S.: GWDL: a graphical workflow definition language for business workflows. In: Gaol, F. (ed.) Recent Progress in Data Engineering and Internet Technology. LNEE, vol. 156, pp. 205–210. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-28807-4\\_29](https://doi.org/10.1007/978-3-642-28807-4_29)
35. Russell, N., Aalst, W.M.P., Hofstede, A.H.M., Edmond, D.: Workflow resource patterns: identification, representation and tool support. In: Pastor, O., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 216–232. Springer, Heidelberg (2005). doi:[10.1007/11431855\\_16](https://doi.org/10.1007/11431855_16)

36. Jacob, F., Gray, J., Wynne, A., Liu, Y., Baker, N.: Domain-specific languages for composing signature discovery workflows. In: Workshop on Domain-Specific Modeling, pp. 61–64. ACM (2012)
37. Alajrami, S., Romanovsky, A., Watson, P., Roth, A.: Towards cloud-based software process modelling and enactment. In: Model-Driven Engineering on and for the Cloud, CloudMDE 14, vol. 1242, pp. 6–15 (2014)
38. Martin, D., Wutke, D., Leymann, F.: A novel approach to decentralized workflow enactment. In: Enterprise Distributed Object Computing, pp. 127–136. IEEE (2008)
39. Johnson, R., Woolf, B.: The type object pattern. In: EuroPLoP (1996)
40. Atkinson, C.: Meta-modelling for distributed object environments. In: Enterprise Distributed Object Computing Workshop, pp. 90–101. IEEE (1997)
41. Rivera, J.E., Ruiz Gonzalez, D., Lopez Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL model transformations. In: Proceedings of MtATL, vol. 9, pp. 34–46 (2009)
42. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the kepler system: research articles. *Concurrency Comput.: Pract. Exp. Workflow Grid Syst.* **18**, 1039–1065 (2006)
43. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Novel features in languages of the epsilon model management platform. In: Modeling in Software Engineering, pp. 69–73. ACM (2008)
44. Kolovos, D.S., Paige, R.F., Polac, F.A., Rose, L.M.: Update Transformations in the Small with the Epsilon Wizard Language. *J. Object Technol.* **6**, 53–69 (2007)

Model-Driven Engineering and Software Development  
4th International Conference, MODELSWARD 2016,  
Rome, Italy, February 19-21, 2016, Revised Selected  
Papers

Hammoudi, S.; Pires, L.F.; Selic, B.; Desfray, P. (Eds.)  
2017, XIV, 355 p. 131 illus., Softcover  
ISBN: 978-3-319-66301-2