Université de Montréal

Using Workflows to Automate Activities in MDE Tools

par
Miguel Gamboa

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en computer science

septembre, 2016

RÉSUMÉ

Le génie logiciel a pour but de créer des outils logiciels qui permettent de résoudre des problèmes particuliers d'une façon facile et efficace. À cet égard, l'ingénierie dirigée par les modèles (IDM), facilite la création d'outils logiciels, en modélisant et transformant systématiquement des modèles. À cette fin, l'IDM s'appuie sur des *workbenches* de langage : des environnements de développement intégré (IDE) pour modéliser des langages, concevoir des modèles, les exécuter et les vérifier. Mais l'utilisation des outils est loin d'être efficace. Les activités de l'IDM typiques, telles que la création d'un langage de domaine dédié ou créer une transformation de modèles, sont des activités complexes qui exigent des opérations souvent répétitives. Par conséquent, le temps de développement augmentate inutilement. Le but de ce mémoire est de proposer une approche qui augmente la productivité des modélisateurs dans leurs activités quotidiennes en automatisant le plus possible les tâches à faire dans les outils IDM. Je propose une solution utilisant l'IDM où l'utilisateur définit un flux de travail qui peut être paramétré lors de l'exécution. Cette solution est implémentée dans un IDE pour la modélisation graphique. À l'aide de deux évaluations empiriques, je montre que la productivité des utilisateurs est augmentée et amééliorée.

Mots clés: Flux de travail, Enactment, Modélisation de domaine dédié, Transformation de modèles, Loi de Fitts.

ABSTRACT

Software engineering aims to create software tools that allow people to solve particular problems in an easy and efficient way. In this regard, Model-driven engineering (MDE) enables to generate software tools, by systematically modeling and transforming models. In order to do this, MDE relies on language workbenches: Integrated Development Environment (IDE) for engineering modeling languages, designing models executing them and verifying them. However, the usability of these tools is far from efficient. Common MDE activities, such as creating a domain-specific language or developing a model transformation, are nontrivial and often require repetitive tasks. This results in unnecessary risings of development time. The goal of this thesis is to increase the productivity of modelers in their daily activities by automating the tasks performed in current MDE tools. I propose an MDE-based solution where the user defines a reusable workflow that can be parameterized at run-time and executed. This solution is implemented in an IDE for graphical modeling. I also performed two empirical evaluations in which the users' productivity is improved.

Keywords: Workflow, Enactment, Domain-specific Modeling, Model Transformation, Fitts Law.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

I would like to dedicate this thesis to many people that were beside me throughout this exciting journey. To my wife, for her endless love, support and encouragement. To my son, who has been a source of inspiration. You have made me stronger, better and more playful than I could have ever imagined. To my parents, thank you for your tenderness, devotion and sacrifice.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof.Eugene Syriani for the continuous support of my study and research, for his patience, motivation, enthusiasm, and immense knowledge.

I thank my fellow labmates in GEODES Group, for the stimulating discussions and for all the fun we have had in the last two years.

Last but not the least, I would like to thank my family: my wife, my son and my parents, for supporting me spiritually throughout my life.

CHAPTER 1

INTRODUCTION

1.1 Context

Software engineering aims to create software tools that allow people to solve particular problems in an easy and efficient way. To this end, once the solution to these problems is found, the following step is to optimize this solution. One particular optimization is to increase productivity during software development.

Model-Driven Engineering (MDE) is a software development approach that promotes automation dealing with domain-specific concepts that abstract away code. [48]. 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 [58], some of the less satisfactory results include the presence of several language workbenches [13]. These are Integrated Development Environments (IDE) to implement DSL, design, transform and verify models. MDE tools and language workbenches, such as AToMPM [56], EMFText [51], GME [30] and MetaEdit+ [22], provide many functionalities, such as DSL creation, model editing, or model transformations development and execution. Although based on common foundational principles, the process for performing these tasks differs greatly depending on the tool used. Each of these tools 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.

1.2 Problem Statement and Thesis Proposition

All activities and tasks in modeling tools require context-dependent decisions leading to an excessive amount 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 to let 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 [51] or scripts [38]. 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 [41], but do not support their execution (or *enactment*) natively. Other executable process languages like BPEL [40] are too generic for the tasks we want to achieve in modeling tools. Workflow languages, such as UML activity diagrams, can be enacted [52], but the execution relies on programming individual actions which hampers porting a process from one tool to another.

Therefore, our proposal is to define an executable workflow that fits exactly the purpose of designing workflows for common tasks in MDE tools. Tasks encompass simple operations, such as opening, closing or saving models, and more complex tasks, such as generating the artifacts of 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 parameterized at runtime 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.

1.3 Contributions

The goal of this thesis is to improve the productivity of modelers using MDE tools by automating repetitive activities. The contributions of this thesis are the following:

1. A language to design and execute workflows that automate common MDE tasks.
2. An empirical analysis of the minimal effort required to perform activities with workflows.
3. An empirical evaluation with real human users thats shows that mechanical efforts are reduced and fewer errors occur when using workflows.

1.4 Outline

This thesis is organized as follows. In Chapter 2, we present relevant information and related work. In Chapter 3, we describe details of our solution and discuss about how we solved challenges faced. Furthermore, we report on the implementation of our approach in AToMPM in a idealistic context. In Chapter 4,we analyze the impact our approach has on improving the user productivity in AToMPM. In Chapter 5, we perform an empirical user study to evaluate the improvement in productivity for real users. In Chapter 6, we improve the workflow language to further automate workflow design and execution. Finally, we conclude in Chapter 7.

CHAPTER 2

STATE OF THE ART

2.1 Model-Driven Engineering

MDE is a software development paradigm that enables to generate software tools, by systematically modeling and transforming models [15]. A model is an abstraction of a real system. Models play on crucial role as they provide information about the structure and behavior of software artifacts.

2.1.1 Modeling

Modeling is a fundamental concept in software engineering and even more in software development. Modern computer systems have reached a complexity that requires us to analyze them at different levels of abstraction. This is where model-based methodologies and solutions ensure an adequate solution. However, diversity in the design process requires several formalisms designed for specific tasks. Intelligent design involves different models of different levels of abstraction which, when combined, maximize the knowledge we have of a system [33]. The future of engineering and software development emphasizes the use of models and the importance of design and implementation, including code generation or model transformation [39].

2.1.2 Domain-specific languages

MDE aims to reduce the gap between problem and software implementation domains through the use of technologies that support transformation of problem level abstractions to software level implementations [15]. In order to achieve this, models describe complex systems at multiple levels of abstraction and MDE has mechanisms to transform

models into running systems. These techniques shield stakeholders from the complexities of underlying implementation technologies. Within MDE, domain-specific modeling focuses on creating models that leverage specific abstractions to a particular domain, as opposed to abstractions in lower level programming languages [23]. Thus, stakeholders can use DSLs to model their problems using abstractions from their own domains of expertise. A DSL is a modeling language tailored to the needs and habits of specific domain experts using notations and concepts they are familiar with.

A DSL is composed of three elements: abstract syntax, concrete syntax and semantics [37]. The abstract syntax defines the main concepts of the language, their relationships and constraints. The concrete syntax defines the language notation to represent and render models, which can be textual or graphical. The semantics of a DSL gives meaning to domain-specific models often by means of model transformation.

2.1.3 Metamodeling

A key element of the abstract syntax is the metamodel of the DSL [26]. It defines types, relations and static semantics of the language. To be well-formed, a model conforms to its metamodel that specifies its permissible syntax. A metamodel is very often represented using UML class diagrams notation [42]. Classes represent the entities of the language. They can contain attributes to retain relevant characteristics of the class. Classes can be related by associations, composition, or specialization relations. Thus pragmatically, a model is an object model instance of the class diagram of the metamodel. As such, models are made of objects, attribute values and links, respectively instantiating classes, attributes and associations from the metamodel.

2.1.4 Deep metamodeling

Standard MDE approaches propose an instantiation mechanism that works as follows: when a model element is instantiated from an metamodel element, the attributes and associations for the metamodel element becomes slots and links of the model element and for this reason, they are not available for further instantiation. If an model element wants to have attributes or associations, these have to be defined explicitly or by specialization[5]. This is a know problem with strict 2-level metamodeling.

One solution is to apply techniques from deep metamodeling [28], and in particular, the approach defining metamodels with potency. The potency of a model element is a number that defines the depth to which a model element can be instantiated. An attribute with potency 0 is a slot holding a value that must be set and can not be further instantiated, attribute with potency 1 must be instantiated at the next meta-level (default in 2-level metamodel-model relationship), and an attribute with potency 2 will be passed along at the instance level as an attribute and will only be assigned a value after two instantiations. There are software tools like Melanee [4] or Metadepth [10] that support deep metamodeling with potency.

2.1.5 Model transformation

In MDE, models can be manipulated using model transformation. It is used for code generation, model validation, model refactoring, translation mappings to produce models, and execution via simulation of models in a systematic way. There are over 20 uses in the model transformation intents catalog [31]. Model transformation is defined at the level of the metamodels but executed at the model level. In general, this operation uses a source model as input and produces a target model as output, where each model conforms to its respective meta-model. Model transformation literature considers a broad range of software development artifacts as potential transformation subjects [9].

A model transformation is made up of patterns, transformation units, and scheduling. Patterns specify the locations in the model where a rule is applied. The precondition pattern must be found in the input model. A left-hand side (LHS) pattern contains those preconditions that must be met before applying the rule. To inhibit its application negative application conditions (NAC) can be used. Furthermore, precondition pattern elements may have specific constraints over their attributes. The postcondition pattern must be found in the output model after the rule is applied. A right-hand side (RHS) pattern contains those postconditions, and actions to be performed on attributes of its pattern elements.

Transformation units complement the rules by adding expressiveness to vary how a rule is applied on a model: e.g., applied once on a single match of the precondition pattern or applied as long as a match is found. Scheduling represents how the rules are executed. Scheduling can be achieved by explicit control structures or can be implicit due to causality dependencies between rules. Typical control structures include sequencing, looping and branching of rules. For example, the model transformation language MoTif [55] offers several transformation units, called rule blocks. An ARule applies a rule on the first match it finds. An FRule applies the rule on all matches found simulataneously. An SRule applies a rule recursively as long as a match is found. In MoTif, rule blocks are scheduled depending on the outcome of the rule, whether it was applied or not. This rule-based graph transformation language is espcially well-suited for defining simulations of a DSL and executing models [55].

2.1.6 Tools

Language workbenches are IDEs for engineering modeling languages, to implement DSL, design, transform, execute and verify models. Two of these tools are described below.

2.1.6.1 AToMPM

AToMPM [56] is an open-source framework for designing DSL environments, performing model transformations, as well as manipulating and managing models. It is a research framework from which one can generate domain-specific modeling web-based tools that run on the cloud. AToMPM uses the most appropriate formalisms and processes, being completely modeled by itself.

To create a DSL in AToMPM [6], 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 icons and relations. He then generates the domain-specific modeling environment by loading the compiler toolbar. Finally, by using the concrete syntax created, the user can define a new model.

2.1.6.2 EMFText

EMFText [12] enables developers to define textual DSLs. Metamodels are described in Ecore, being implemented in the Eclipse Modeling Framework (EMF). To create a DSL in EMFText 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. Appendix VI has in detail the steps to create a DSL in AToMPM and EMFText.

The are important differences between these two tools, AToMPM can create graphical DSL while EMFtext supports textual DSL. The concrete syntax in EMFtext is repre-

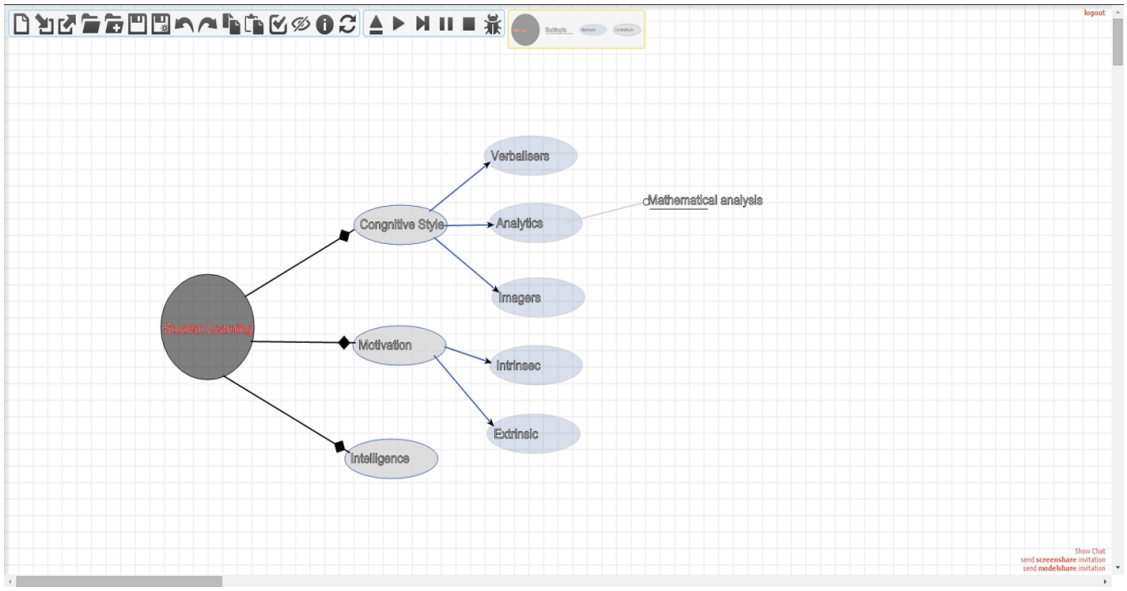


Figure 2.1 – AToMPM user interface showing a domain-specific model

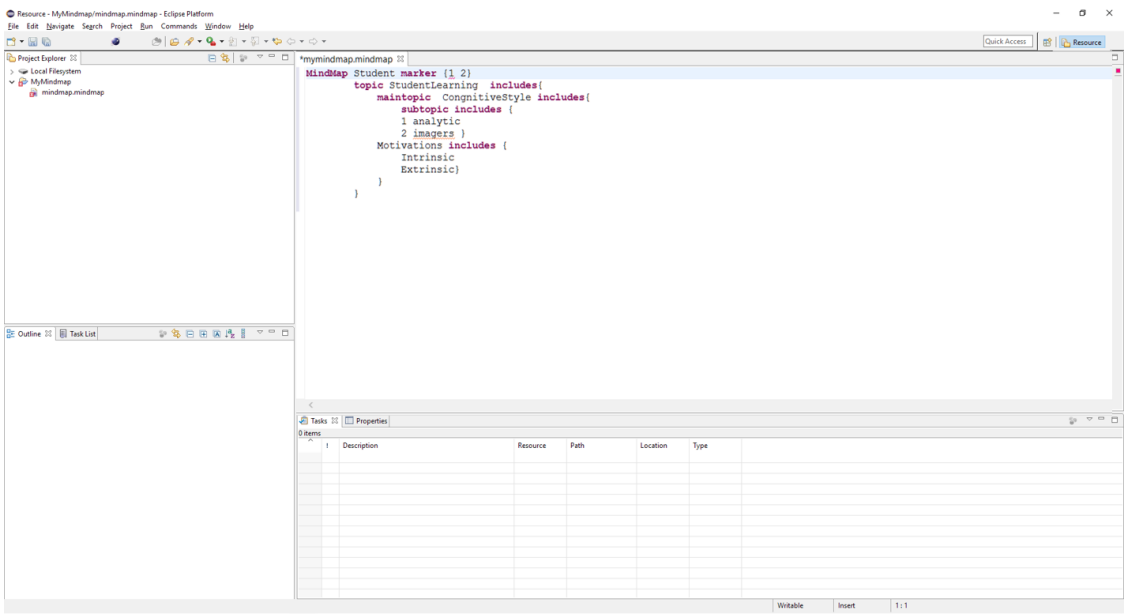


Figure 2.2 – EMFText user interface showing a domain-specific model

sented through a grammar, while in ATOMPM is a set of icons and graphs.

Observing the process of DSL's creation in both tools, we note that modelers have to know all the steps and perform many tasks and user interactions because the processes to follow are complex for all users, whether they are language engineers or domain-specific modelers, since they require heavy mental loads and tasks that are error-prone. Therefore, it is necessary to automate parts of the process to improve the modeling process.

2.2 User Errors in Tools with Graphical Interaction

Many of MDE activities such as DSL creation, model editing, or model transformations 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. In the end, users are spending more time on development than necessary.

Type of errors

As [27] defines, a system works fine when his functionalities do what they must do. A failure, is an event that occurs when the functionalities deviates from proper functioning. A system fails either because it does not comply with the functional specification, or because this specification did not adequately describe the system function. It is an *error*.

A good classification for the types of errors in Graphical User Interface (GUI) is found in [29], if the user makes an unnecessary action in performing the current task this is, in most cases, an error. However, the user may have wanted to go backwards in the interaction to a previous step. Another common type of error is an action performed belonging to the task, but the user has failed to do some necessary actions before. An error could also arise if the user inputs something to the program that is not correct.

Based on the above, the following four categories or of errors types were created:

Typographic error

A typographical error occurs when the user enters erroneous information into the system using the keyboard . For example entering the wrong name of a file or its extension.

Functional error

A functional error is an error in which the user makes a mistake in the proper functioning of the system. For example, clicking on the wrong button when you want to perform a specific action. The desired action is not achieved since he clicked on another button.

Preconditional error

This mistake is made when a series of steps are needed to complete an action and one of these steps is omitted. This leads to not being able to perform the desired action.

Backtracking error

This error occurs when the user goes back to a previous step without completing a process.

2.3 Workflows

A workflow is the study of the operational aspects of a work activity: how tasks are structured, how they perform, how they are synchronized, how information flows to support the tasks and how monitoring is done to compliance tasks. Workflows have

been used to support various types of business processes [18]. The workflow patterns initiative was established aiming to delineate the fundamental requirements that arise during business process modeling [47].

The Basic Control Flow Patterns captures elementary aspects of process control. The following briefly explains each one.

2.3.1 Sequence

The sequence pattern serves as the fundamental building block for workflows. It is used to construct a series of consecutive tasks which execute in turn one after the other. Two tasks form part of a sequence if there is a control-flow edge from one of them to the next which has no guards or conditions associated with it.

2.3.2 Parallel split

The parallel split pattern allows a single thread of execution to be split into two or more branches that can execute tasks concurrently. These branches may or may not be resynchronized in the future.

2.3.3 Synchronization

Synchronization provides means of reconverting the execution threads of two or more parallel branches into a single one. In general, these branches are created using the parallel split construct earlier in the process model. The thread of control is passed to the task immediately following the synchronizer once all of the incoming branches have completed.

2.3.4 Exclusive choice

The exclusive choice pattern allows the thread of control to be directed to a specific (subsequent) task depending on the outcome of a preceding task, the values of elements of specific data elements in the process, the results of an expression evaluation or some other form of programmatic selection mechanism. The routing decision is made dynamically allowing it to be deferred to the latest possible moment at runtime.

2.3.5 Simple merge

The simple merge pattern provides a mean of merging two or more distinct branches without synchronizing them. As such, this presents the opportunity to simplify a process model by removing the need to explicitly replicate a sequence of tasks that is common to two or more branches. Instead, these branches can be joined with a simple merge construct and the common set of tasks needed only to be depicted once in the process model.

2.3.6 Arbitrary cycles

The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches.

2.3.7 Structured loop

There are two general forms of this pattern: the while loop which equates to the classic while...do and the repeat loop which equates to the repeat...until construct.

2.3.8 Transient trigger

Transient triggers are a common means of signaling that a predefined event has occurred and that an appropriate handling response should be undertaken.

2.3.9 Persistent trigger

The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task

2.4 Existing Approaches to Automate User Activities

A lot of work can be found in the literature on workflow definition and enactment [35, 46, 60]. In [19], 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 [2, 36]. However, none of these approaches models workflow enactment explicitly as we did using model transformation. We propose 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 [21] where both types and instances are explicitly modeled in the Metamodel. This is similar to the notion of clabject [3] which generalizes this approach.

Existing works use transformations chains, but not workflows. Others approaches execute wizards to automate repetitive tasks, but none combine workflow definition, workflow execution and MDE techniques. This makes our approach unique. With these approaches, no one reported an improvement performing task.

2.4.1 The FTG+PM language

From an implementation point of view, the closest work to ours automates transformation chains in AToMPM [32]. The FTG+PM language is defined using two sub-languages: the Formalism Transformation Graph (FTG) language and a Process Model (PM) language. 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 building blocks of the FTG are formalisms (nodes in the graph) and transformations (edges in the graph). The FTG describes the different languages that can be used at each stage of model development. The transformations model development activities, and the control flow and data flow between each transformation action are explicitly modeled in the PM. 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. The transformations defined as activities in the PM are all modeled as rule-based graph transformations using AToMPM's transformation language. 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.

2.4.2 Wires

Wires [44] is a graphical executable language for orchestrating ATL transformations. Wires assumes a data-flow process, in which a set of input models (conforming to their corresponding metamodels) are processed by a chain of ATL transformations until a set of output models is produced. Similarly to FTG+PM, Wire supports the specification and execution of model transformation workflows. Basically, Wires provides mechanisms to create model transformations chains. The chain is composed of transformations, which act as processing nodes. Parameters represent the consumed and produced data

by transformations. Transformations are wired together by directed connectors (Wires) that indicate how the outputs of the transformations are linked to the inputs of the next ones.

2.4.3 Epsilon wizard language

The Epsilon Wizard Language (EWL) [24] provides tailored and effective support for defining and executing update transformations on models of diverse metamodels. Severals tools provide built-in transformations (wizards) for automating common repetitive tasks. However, according to the architecture of the designed system and the specific problem domain, additional repetitive tasks typically appear, which cannot be addressed by the preconceived built-in wizards of a modeling tool. EWL helps to create wizards for those specific needs.

In our approach, activities essentially encapsulate model management tasks. The Epsilon language suite [25] 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.

EWL whose purpose is to refactor, refine, and update models 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 pop-up 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.

CHAPTER 3

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 chapter, 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.

3.1 Language for Semi-Automated Workflows

We model the DSL for defining activities that can be performed in MDE tools. An activity is composed of tasks, to define concrete actions to be performed, and control nodes, to define the flow of tasks. The metamodel in Figure 3.1 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.

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 user's activities as much as possible, therefore most of the tasks are automatic: they do not require human interaction. For example,

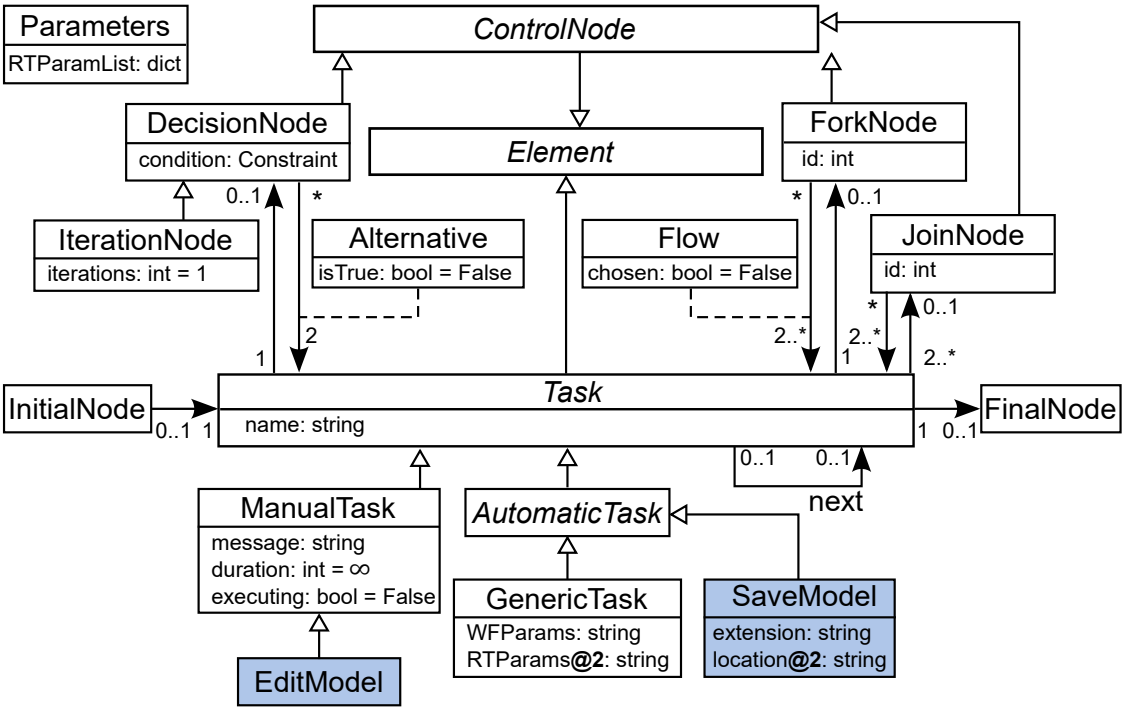


Figure 3.1 – Generic metamodel of workflows for modeling tools

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 Figure 3.1 (`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 activity.

A workflow conforming to the 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 depending on a Boolean condition evaluated at run-

time. Repetitions are possible with an iteration node. 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. These correspond to the common basic control flow patterns for workflows [47]. Although not supported in our current implementation, tasks may be executed concurrently, except if the concurrent tasks are manual.

3.2 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.

3.3 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 [16], 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 [28], and in particular, the approach defining metamodels with potency. 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 Figure 3.1. 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 Figure 3.1 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 [59].

3.4 Workflow Enactment by Model Transformation

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

3.4.1 Deep instantiation

The issue with the above solution is that not many modeling frameworks(e.g., AToMPM [1] and EMF) support deep metamodeling with potency like Metadepth or Melanee 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 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

1. In [57], 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.

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 3.2 depicts the transformation in MoTif [55], a rule-based graph transformation language in AToMPM. Rules are defined with a precondition pattern on the left and a post-condition pattern on the right. Constraints `Const` and actions `Act` on attributes are specified in Python. The transformation in Figure 3.2 populates all attribute fields of the parameter object (the icon with two gears) by visiting each task in the activity model. The attributes names and types are stored in a JSON format that is then used to render a wizard prompting for their corresponding values to the user. This is performed in a single FRule that makes sure that each task is visit exactly once. Note that the transformation uses a FRule to make sure that each task is visited exactly once, which is why no negative application condition is needed.

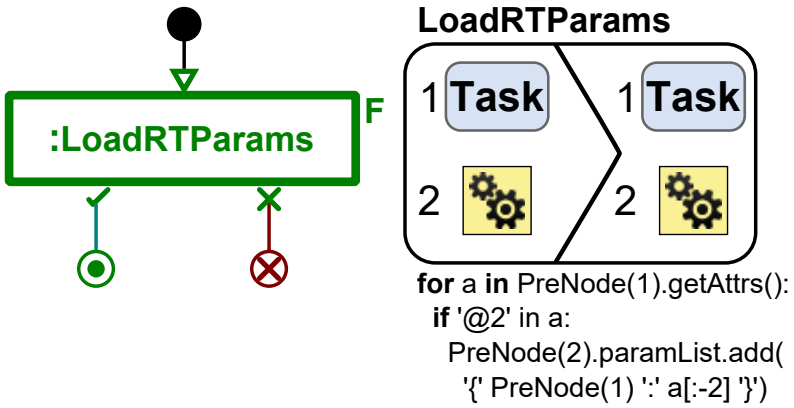


Figure 3.2 – Transformation for loading run-time parameters in MoTif

3.4.2 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 [32]. 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 3.3 illustrates the overall structure of this transformation and Figure 3.4 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 precondition. 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 interacting 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

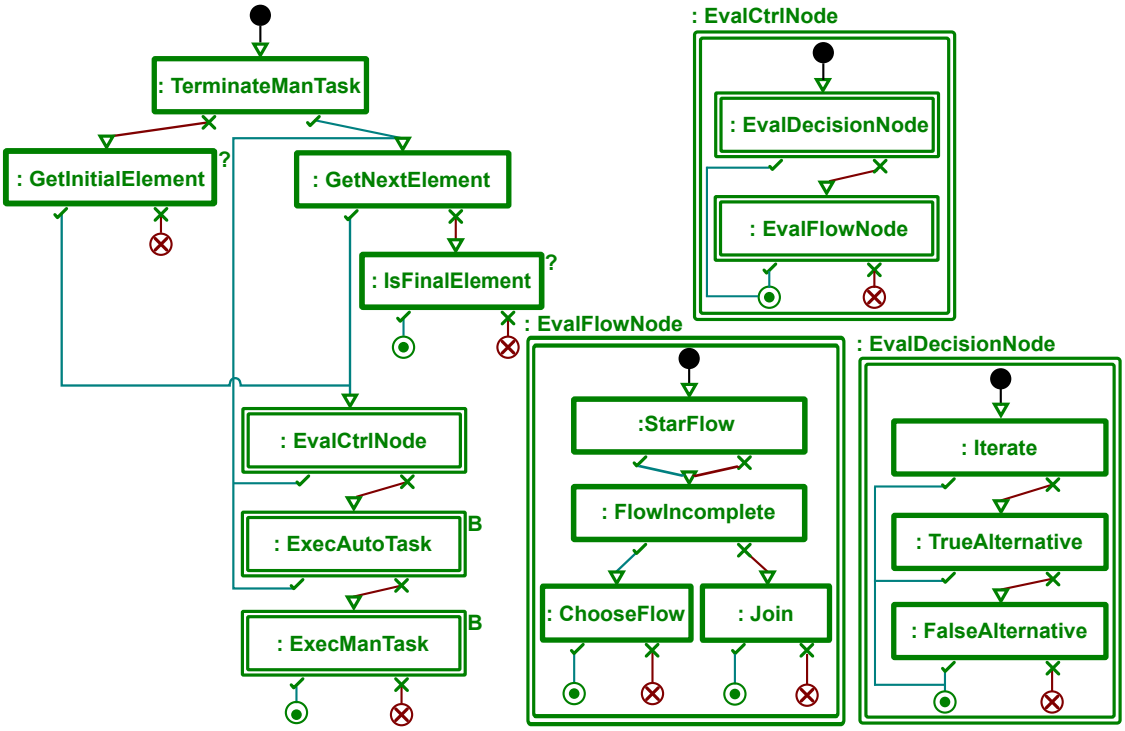


Figure 3.3 – Control structure of the transformation in MoTif that executes a workflow

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 [54], most do not. Therefore, as depicted in Figure 3.3, we extend the logic to handle manual

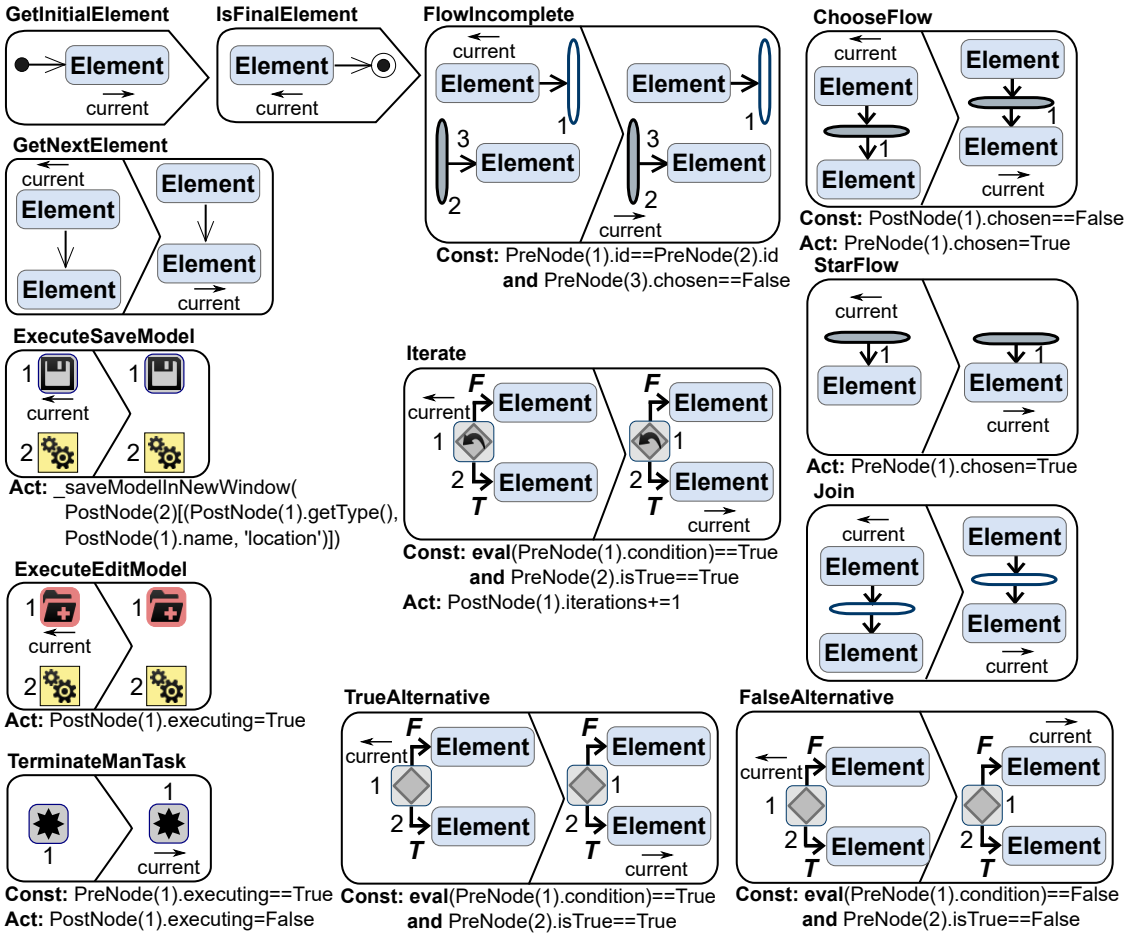


Figure 3.4 – Transformation rules in MoTif that execute a workflow

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.

3.5 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 Figure 3.1 and add a rule under `ExecAutoTask` or `ExecManTask` in Figure 3.3 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 [45, 53], 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 Figure 3.3. The user must then manually recover from the error and restart the execution of the workflow. Nevertheless, run-time parameters are retained.

3.6 Implementation in AToMPM

We implemented a prototype in the MDE tool AToMPM [56], 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 Figure 3.1. Figure 3.5 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 (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 Figure 3.5. All these operations can be automated, since they require a location as run-time parameter. `SaveModel` also has

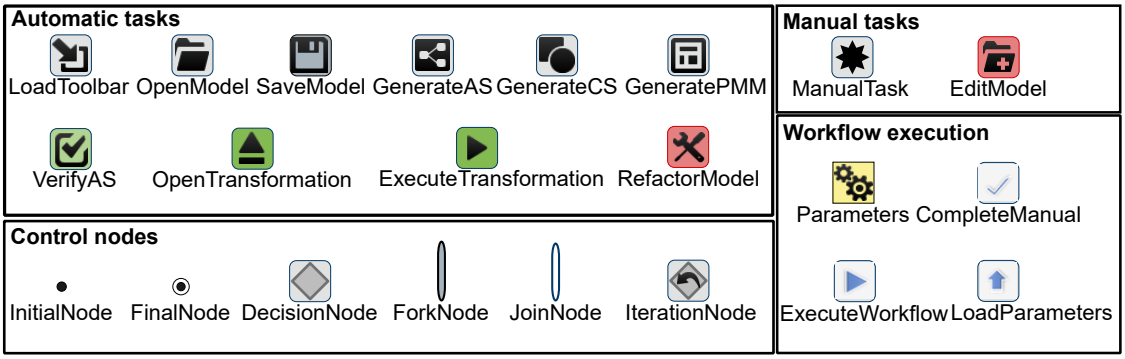


Figure 3.5 – Concrete syntax of the workflow DSL in AToMPM

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.

3.7 Process

Our prototype is to be used as follows:

1. The designer defines workflows by creating instances of the workflow DSL.
2. A user (a language engineer in this example) then selects which workflow he desires to enact; in this case, a DSL workflow.
3. 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 Figure 3.2. This is shown in Figure 3.6
4. The user push OK button as seen in Figure 3.7 .
5. Upon pushing `ExecuteWorkflow` button shown in Figure 3.8 , the simulation (presented in Figure 3.3) executes the workflow autonomously. When a manual task is reached, a new AToMPM window is opened with all necessary toolbars preloaded. A message describing the manual task to perform is displayed to the user and the simulation stops.
6. After the user completes the task, he pushes the `CompleteManual` button. Then, the window closes and the simulation restarts. This is shown in Figure 3.9

3.8 Example Workflow for Creating a DSL

Figure 6.3 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

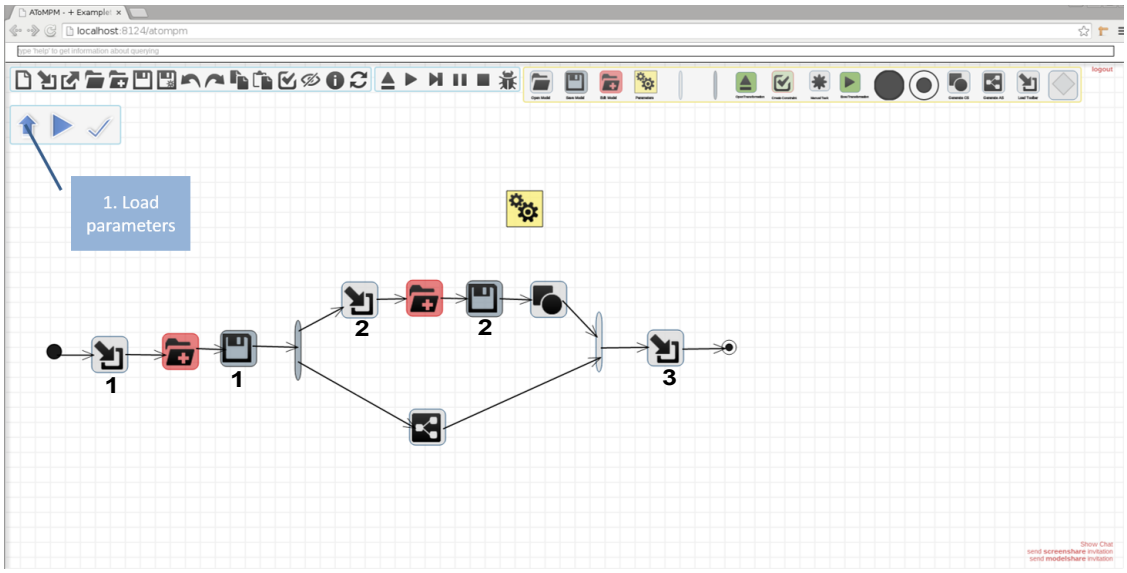


Figure 3.6 – Step 1 to create a DSL using workflows. Load Parameters

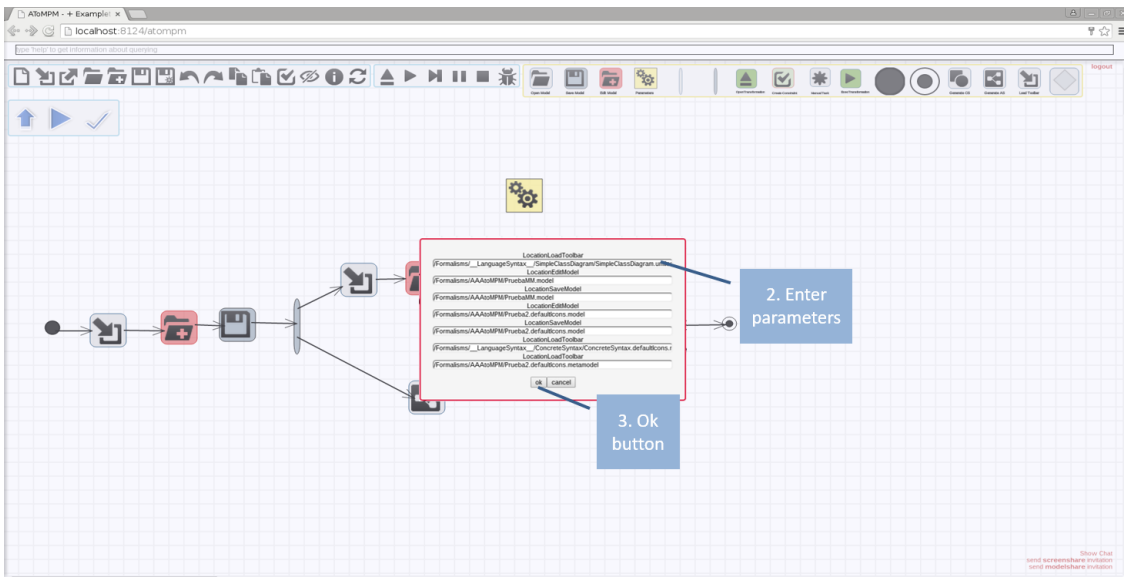


Figure 3.7 – Step 2 to create a DSL using workflows. Enter the Parameters

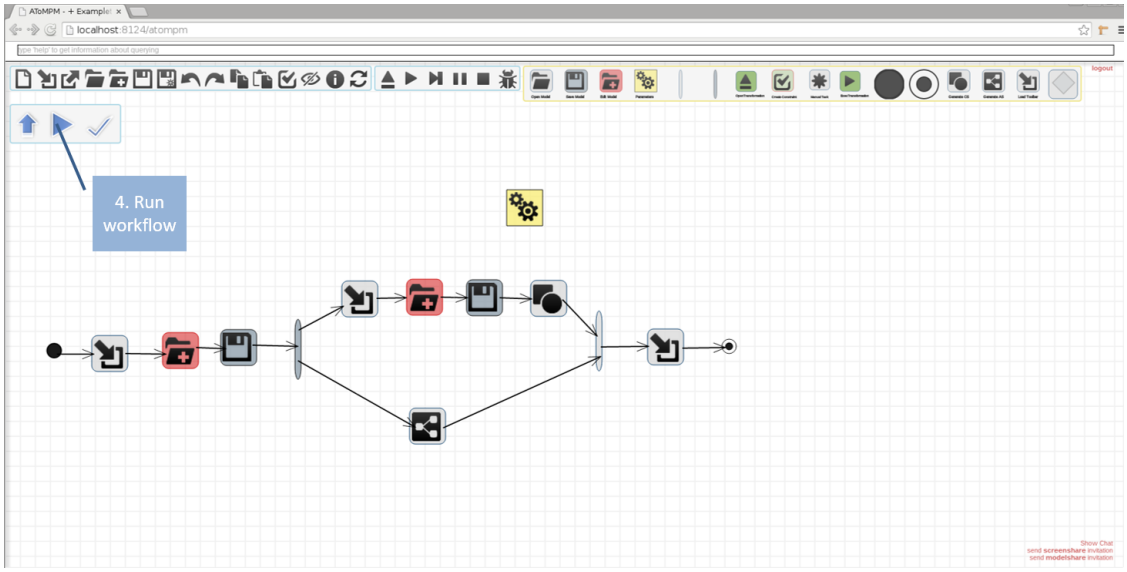


Figure 3.8 – Step 3 to create a DSL using workflows. Enact a workflow

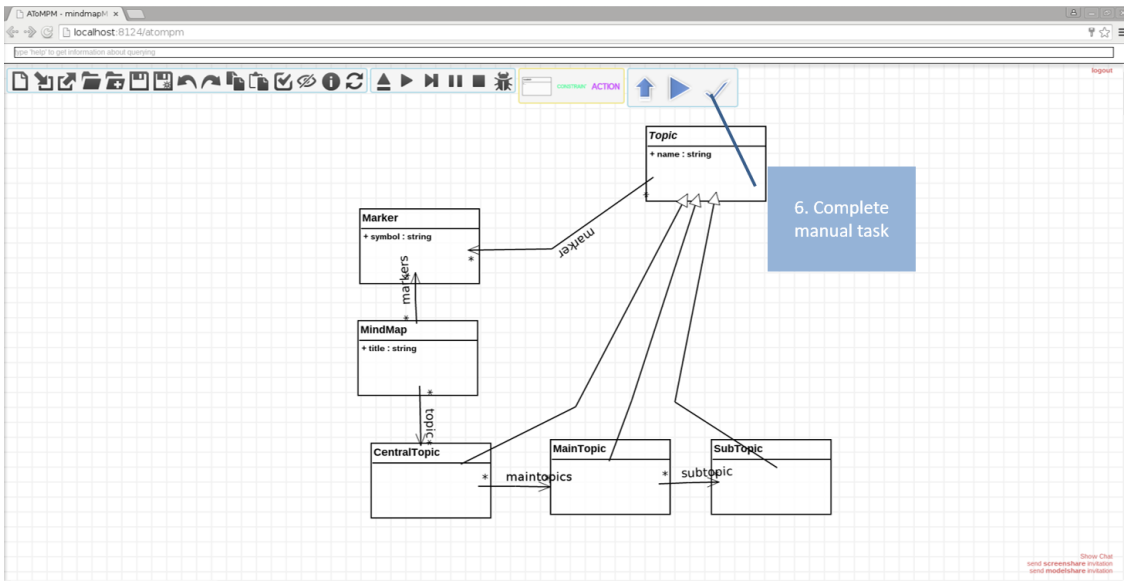


Figure 3.9 – Step 4 to create a DSL using workflows. Complete a manual task

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 `SaveModel`. 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. 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.

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`.

When the join node is reached again, this time all flows were executed and proceeds with the final task `LoadToolbar`. The simulation ends on a new window open with the new DSL loaded, ready for the user to create his domain-specific model.

CHAPTER 4

ANALYSIS OF THE IMPROVEMENT WHEN USING WORKFLOWS

We perform an evaluation of the impact our approach has on improving the user productivity in AToMPM. However, this study does not rely on users and the results are independent from user performance. To do so, we focus on the mechanical efforts user would need to complete an activity. This corresponds to mouse operations for a graphical MDE tools like AToMPM. We extrapolate the cognitive efforts of the user to be the delays between individual mechanical operations.

4.1 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.

4.2 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 [34] 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. T_{FL} is calculated using the formula in

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 [17], the average time to click reactively is 258 milliseconds. 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 K of an activity is the minimum numbers of clicks it requires the user to perform. In this case, the minimum number of clicks required to complete an activity equals the number of clicks C, since this study was designed in a scenario without errors where the number of clicks an activity was optimized. Hence $K = C$.

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.

4.3 Experimental Setup

We performed all experiments on a 15.6" laptop monitor with a resolution of 1920×1080 pixels. The machine was an ArchLinux virtual machine using 2 cores and 4GB of RAM, running on Windows 10 quad-core computer at 2.4 GHz with 16 GB 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. The software used to collect data are: Page Ruler [43] and Perfect Screen Ruler [49] to measure several distances and Auto Mouse Clicker to record mouse movements and clicks.

4.4 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%.

These results lead us to the following equation that allows us to predict the mechanical time for a given screen size and distance of objects and AToMPM. The time of mouse movement for AToMPM is

$$T_{FL} = 166.75 + 155.93 \times log_2(1 + D/S) \tag{4.1}$$

In our prototype, we implemented the five most common tasks in AToMPM shown in Figure 3.5. 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.

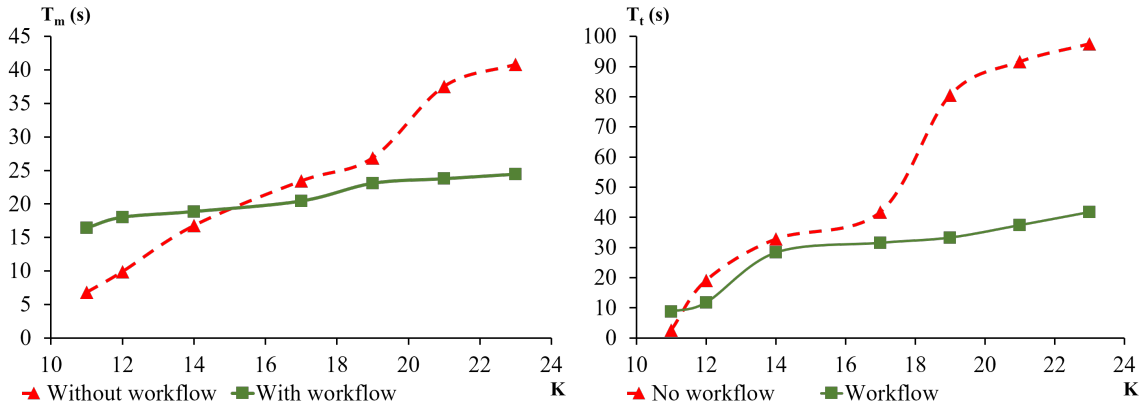
The most complex activity we evaluated is for the creation of a DSL in AToMPM modeled with the workflow in Figure 6.3, consisting of seven automatic tasks. The workflow starts by loading the `Class Diagram` formalism. It lets the user manually create the appropriate class diagram model to define the 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.

4.5 Results

The two plots in Figure 4.1 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.

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 Figure 4.1(a), is due to the overhead to open the appropriate workflow and set all run-time parameters. The reason why T_m plateaus after $K = 17$ 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 4.1(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 $K > 11$ 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 $K > 12$, the gap remains almost identical. When performing the experiments, we noted



(a) (b)

Figure 4.1 – Mechanical (a) and cognitive (b) efforts with respect to the number of tasks in a workflow

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 4.I 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.

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.

	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%

Table 4.I – Time measurements in seconds and improvements when using workflows for $K = 23$ task complexity

4.6 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 preprocessed 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 knowl-

edge 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.

CHAPTER 5

USER STUDY TO EVALUATE THE IMPROVEMENT IN PRODUCTIVITY WHEN USING WORKFLOWS

The analysis in Chapter 4 assumed perfect users, namely, users who perform mouse movements crossing the shortest distance possible, who do not make any user interaction error and who use the minimal number of clicks to complete MDE activities. In this chapter, the goal is to take into consideration the human factor to have more realistic results. We evaluate the improvement of productivity emphasizing on their performance, measuring real mechanical efforts, number of errors, and effort to correct errors when completing MDE activities.

5.1 Research Questions

In this user study, we are specifically interested in answering the followings research questions:

RQ1: "Do workflows improve productivity by reducing the development time for real users without errors in automated task?". Although this was answered in the analysis of Chapter 4, we want to validate the results with real users. For this question, we focus on automated tasks only and do not take errors into account.

RQ2:"Do workflows reduce the number of errors when performing an activity?". We are in particular interested if, during manual tasks, the amount of errors decreases when the user performs activities using workflows. This is because these tasks make him focus on core MDE tasks (such as creating a metamodel) rather than operating the tool (such as saving a model). This question will be answered for both automatic and manual tasks.

RQ3: "Do workflows improve corrective efforts when automating activities?". Having observed that the user makes errors when focusing on core tasks, we are interested in analyzing the effort required to correct them.

5.2 Study Design

To answer the research questions, we conducted a user study with a set of experiments in a controlled environment.

5.2.1 Experimental setup

All the experiments were performed on a 24 inch monitor with a resolution of 1920×1080 pixels. The virtual machine was the same as the one we was used in Chapter 4, running on a Windows 10 dual-core computer at 2.4 GHz with 16 GB of RAM. No time limit was imposed, but participants each took between one to three hours to complete the experiment. All experiments were performed in a dedicated isolated closed room. The PC was ready to be used with all necessary software and resources installed on it. The available resources were: online tutorials on using AToMPM (found on the AToMPM website [1]), a tutorial on how to use workflows in AToMPM (available in Appendix I), and the directives (available in Appendix II). While conducting the experiments, I was physically present in the room to take notes and answer questions only about the directives and resources, not about the solution itself.

5.2.2 Participant selection

A non-probability sampling was used to select the possible subjects. In this case, it was a convenience sampling. We required experienced users of AToMPM in order to reduce learning time of the tool and shorten the experiments duration. During the study,

1. `http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm`

each participant was met individually and selected from people who have experience in developing DSLs and model transformations in AToMPM. They were all graduate students who had followed an advanced course in MDE and where MDE is their core research subject. To remove any bias or familiarity effect, participants were divided into two groups:

— *Group A* first completed all activities by hand without using workflows and then completed the same activities using workflows.

— *Group B* first completed all activities using workflows and then completed the same activities by hand without workflows.

In total, seven developers participated in this study. The distribution of participants was 4 for group A and 3 for group B.

5.2.3 Activities

Each experiment consisted of performing three typical MDE activities in AToMPM in a specific order. They were all using a common running example of a DSL for mind maps. They first had to generate a domain-specific editor for mind maps and create a model. Then they had to develop an inplace model transformation and run it on the model. Finally they had to modify the initial metamodel by adding a constraint and verify if the model satisfies it. The complete description of the directives is available in Appendix II.

5.2.3.1 Mind maps

A mind map is a diagram used to visually organize information [7]. It is often created around a single concept, the central topic, drawn as an image in the center of a blank page, to which associated representations of ideas such as images, words and parts of words are added. Major ideas are connected directly to the central concept and other

ideas branch out from those. These are the subtopics.

5.2.3.2 Activity: DSL

In this activity, the participant is asked to create a simple DSL for mind maps. The instructions to follow in order are below:

1. Create the metamodel for mind maps provided in the directives.
2. Save this metamodel at a specific location and name it.
3. Generate the abstract syntax from this metamodel.
4. Open a new window.
5. Create the concrete syntax model provided in the directives.
6. Save this concrete syntax model at a specific location and name it.
7. Generate a modeling environment from this concrete syntax model.
8. Open a new window.
9. Load the toolbar that was just generated from step 7.
10. Create a mind map model provided in the directives.
11. Save this model at a specific location and name it.

Figure 5.1 shows the workflow of this activity. The first manual task is to create the metamodel for mind maps using class diagrams. The second manual task is to design the icon model as the concrete graphical syntax of the DSL. The last manual task is to create a mind map model instance of this DSL.

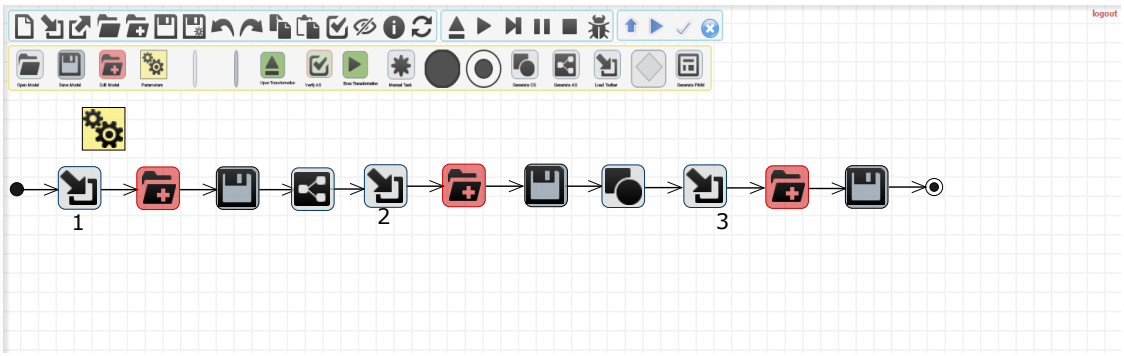


Figure 5.1 – Activity Create a DSL

5.2.3.3 Activity: Transformation

In this activity, the participant is asked to develop an inplace model transformation in MoTif that assigns orders to subtopics of a mind map. The instructions to follow in order are below:

1. In a new window, generate a modeling environment for rule patterns from the mindmap metamodel.
2. Create the first rule provided in the directives, This rule assigns a number to each subtopic.
3. Save this rule model at a specific location and name it.
4. Open a new window.
5. Create the second given rule provided in the directives. This rule orders subtopic uniquely.
6. Save this rule model at a specific location and name it.
7. Open a new window.
8. Create the transformation model to schedule rules provided in the directives.
9. Save this transformation model at a specific location and name it.
10. Open a new window.
11. Load the model created previously at step 10 of the DSL activity.
12. Load the transformation model created at step 8.
13. Execute the transformation.

Figure 5.2 shows the workflow of this activity. In the first two manual tasks, the participant designs the two rules of the transformation. The third manual task is to design the scheduling model that specifies the order of execution of the rules.

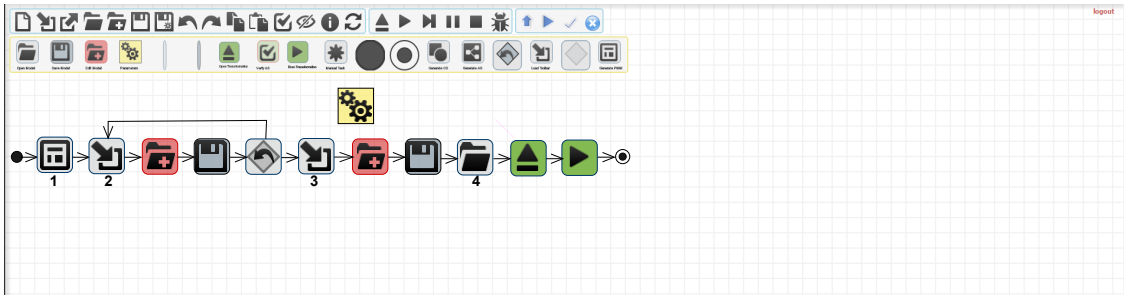


Figure 5.2 – Activity Create a Transformation

The parameters that we will enter are: The location to model to create the pattern model, the location of MoTif toolbar, the location of toolbar for rule creation and the location of transformation previously created.

When the user press `ExecuteActivity` button, the simulation (presented in Figure 3.3) executes the workflow. When an iteration node is reached, the next task is indicated by the black arrow with the number two. The cycle ends when the specified number of iterations, in this case two, is reached and the simulation continues. In this example, there are two iterations conclusions, that correspond to the number of rules to create a transformation.

5.2.3.4 Activity: Evolution

In this activity, the participant is asked to add a constraint to the metamodel that ensures exactly one main topic presented in a mind map model. The instructions to follow in order are below:

1. In a new window, open the mindmap metamodel created at step 1 of the DSL activity.
2. Add the constraint provided in the directives to the metamodel.
3. Save the metamodel at its current location with the same name.
4. Generate the abstract syntax from this metamodel.
5. Open a new window.
6. Load the model that was created previously at step 10 of the DSL activity.
7. Make a specific modification to the model provided in the directives.
8. Check if the constraint is violated.

Figure 5.3 shows the workflow of this activity. The first manual task is to specify the constraint. The second manual task is to add an additional central topic to the model in order to violate the constraint.

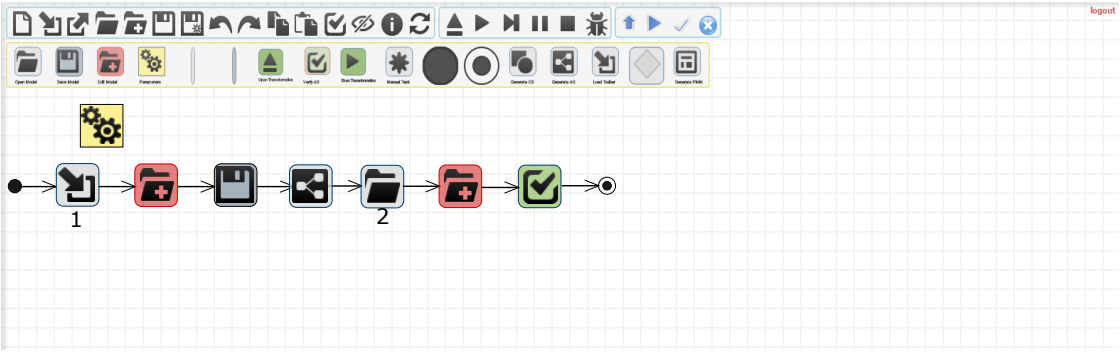


Figure 5.3 – Activity Modify a Metamodel

5.2.4 Feedback survey

At the end of the experiment, participants were invited to complete an online survey. The survey consisted of seven questions. The questions asked for their opinions on using workflows, to rate the usefulness of workflows, and about the experiment setup itself. The goal of this survey is to collect subjective evaluations of our approach and have feedback about the solution. The complete survey is available in Appendix III.

5.3 Metrics

In order to quantitatively analyze this user study, we relied on several metrics that extend those used in Chapter 4.

5.3.1 Independent variables

The independent variables of this study are the following. *Act* defines the activity with one of the three values: *DSL*, *Trafo*, or *Evol* for the respective activities. *Method* defines the approach used to complete each activity: either using workflows *W* or not (i.e., by hand) *H*. Finally, *Task* defines the type of task performed, which is either *Automatic* or *Manual*.

5.3.2 Dependent variables

The dependent variables are similar to those used Section 4.2. We recorded the total time duration T spent by a participant to perform automatic tasks, manual tasks, and the complete activity. T_c is the time to perform a task correctly without errors and T_e the time spent from error occurs until it is corrected. Thus $T = T_c + T_e$. Time is measured in seconds. We also counted the number of clicks C he needed per activity. C_c is the correct number of clicks and C_e the number of clicks in errors. Thus $C = C_c + C_e$.

Additionally, we collected time durations and number of clicks for each error that occurred. Errors were classified by type (cf. Section 2.2): E_{et}, T_{et} and C_{et} denote respectively the number of errors, duration in seconds and number of clicks for typographic errors, E_{ef}, T_{ef} and C_{ef} for functional errors, E_{ep}, T_{ep} and C_{ep} for pre-conditional errors, and E_{eb}, T_{eb} and C_{eb} for backtracking errors. We also denote their respective sums by $E_e = E_{et} + E_{ef} + E_{ep} + E_{eb}$, $T_e = T_{et} + T_{ef} + T_{ep} + T_{eb}$, and $C_e = C_{et} + C_{ef} + C_{ep} + C_{eb}$. Note that error durations and number of clicks start when the error occurs and ends when the error is corrected. Thus they are used to measure the corrective effort.

We did not take into account the number of clicks during manual activities.

Finally, the activity complexity $K(Act)$ denotes the minimum number of clicks to perform one activity.

5.3.3 Revision of activity complexity

As opposed to the previous analysis, in this study, C is not the minimal number of clicks required to perform an activity anymore, since real users may make more clicks than optimally required. We therefore need to revise K to be independent from real user clicks. For this, we consider an activity to be partitioned into automated and manual tasks, where each task has a set of run-time parameters. We denote $P(Act)$ the number of run-time parameters of all tasks in an activity. We also denote $M(Act)$ the number of

manual task in the activity. Then, we define the complexity of an activity as

$$K(Act) = k_0 + k_a \times P(Act) + k_m \times M(Act) \tag{5.1}$$

Here, k_0 is the constant number of clicks to setup the workflow (e.g., opening the workflow model and loading the parameters). k_a is the number of clicks to set the value of a single run-time parameter and k_m is the number of clicks needed to process a manual task (e.g., setting the task as completed, resuming the workflow). When there is a loop $P(Act)$ and $M(Act)$ will be multiplied for the number of iterations In our implementation, the complexity of an activity is given by:

$$K(Act) = 7 + 2P(Act) + 2M(Act) \tag{5.2}$$

Therefore, the complexity of the activities in this user study are: $K(Evol) = 15$, $K(DSL) = 20$, $K(Trafo) = 25$

This metric does not take into account the effort to perform manual tasks because it depends on the problem at hand. Nevertheless, $K(Act)$ gives a satisfactory partial order to compare different activities by their complexity.

5.4 Data Collection

At the end of each experiment, we created a video file that captures all screen activity as well as the audio of the conversations. We obtained prior consent of each participant. We recorded every mechanical action: mouse movements, types of clicks, keystrokes, and window switching. We relied on the software Advance Key and Mouse Recorder [1] and CamStudio [8]. After all experiments were over, we analyzed the videos and collected relevant information from reports generated by the software.

To facilitate and better interpret the data, we were taking notes during the experiment itself. Notes pointed out important events and their time stamp. For example, we noted time and cause when a distraction occurs. This enabled us to discard the metrics of this event when calculating our variables. Also we noted all errors, their causes and their correction to facilitate the analysis of the videos and get the data in the best way possible.

We statistically analyzed the collected data for each metric using SPSS [50] software. Furthermore, we discovered one outlier situation, using a variability coefficient and the modified Thompson Tau test, an objective method to determine if a data point is an outlier. We therefore discarded all the data from the experiment of this participant who took twice the time to complete the experiment compared to the average and who spent a lot of time correcting errors because he misunderstood the directives.

5.5 Analysis of the Results

All the data collected are available in Appendix IV and V. In order to answer the three research questions, we need to validate the following:

5.5.1 Improve productivity by reducing the development time without errors

H_0: There is no difference in the time T_c and number of clicks C_c for mechanical and cognitive efforts to perform a task correctly without errors for method W than for method H.

H_1: The time and number of clicks for mechanical and cognitive efforts to perform a task correctly without errors is reduced when automating activities using method W.

First of all, it is necessary to determine the distribution of the data. We applied a test of normality using the Shapiro-Wilk method on our dependent variables. Figure 5.4 shows the results for the sum variables. The total time T to complete an activity is normally distributed having a significance value greater than 0.05, but the time T_c to perform

Tests of Normality

Method		Shapiro-Wilk		
		Statistic	df	Sig.
T	H	,923	18	,145
	W	,936	18	,244
T_e	H	,903	18	,066
	W	,882	18	,029
C_e	H	,959	18	,591
	W	,808	18	,002
E	H	,770	18	,001
	W	,914	18	,103
C	H	,842	18	,006
	W	,856	18	,010
C_c	H	,777	18	,001
	W	,801	18	,002
T_c	H	,774	18	,001
	W	,818	18	,003

Figure 5.4 – Test of normality, rows in gray indicates normally distributed variables and the whites are not.

a task correctly without errors is not. The elapsed time T_e between the occurrence of an error until its correction not using workflows is normally distributed, nevertheless this time using workflows is not. With regards to the number of clicks, the number of total clicks C needed per activity and the correct number of clicks C_c are normally distributed. Interestingly, the number of errors E using workflows and the number of clicks in errors C_e not using workflows (i.e., by hand) are normally distributed. However, the number of errors not using workflows and the number of clicks in errors using workflows are not normally distributed.

For hypothesis H_0, we need to determine if the results using method W and method H are consistent with each other. Figure 5.5 reports the result of a Mann-Whitney U Test for the variables T_c and C_c. Looking at the significance (in the third column), we observe that all p-values are smaller than 0.05. Therefore, we can reject the null hypothesis and

Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of T_e is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,001[1]	Reject the null hypothesis.
2	The distribution of C_e is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,000[1]	Reject the null hypothesis.
3	The distribution of E is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,004[1]	Reject the null hypothesis.
4	The distribution of C is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,000[1]	Reject the null hypothesis.
5	The distribution of C_c is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,000[1]	Reject the null hypothesis.
6	The distribution of T_c is the same across categories of Method.	Independent-Samples Mann-Whitney U Test	,011[1]	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is ,05.

[1]Exact significance is displayed for this test.

Figure 5.5 – Mann-Whitney U Test for non-parametric variables

we can state that the mechanical effort to perform an activity is improved with workflows when no errors one made.

5.5.2 Reducing the number of errors

H_0: There is no difference in the number of errors E when performing a task using method W or method H.

H_1: The number of errors E is reduced when automating activities using method W.

For E for the method H, we also applied the Mann-Whitney U Test reported in Figure 5.5. Looking at the significance (in the third column), we again observe that p-value is smaller than 0.05. Therefore, we can reject the null hypothesis and we conclude that the number of errors E is reduced when using workflows. This reduction can observed in Figure 5.7(a) where the improvement in number of errors is higher than 60% for each activity.

5.5.3 Using workflows improves corrective efforts

H_0: There is no difference in the time T_e and number of clicks C_e when errors occurs to perform a task for method W than for method H.

H_1: The time and number of clicks when errors occurs is reduced when automating activities using method W.

In Chapter 4 we showed that using workflows improves the time spent on non essen-

Independent Samples Test

		t-test for Equality of Means						
		t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
							Lower	Upper
T	Equal variances assumed	-,035	34	,972	-6,38889	182,21009	-376,68435	363,90657
	Equal variances not assumed	-,035	34,000	,972	-6,38889	182,21009	-376,68435	363,90657

Figure 5.6 – T-test for normally distributed variables

tial activities in a perfect scenario with ideal users. The plots in Figure 5.7 demonstrate a clear improvement in the number of clicks C_e and time T_e to correct errors, not only for automatic task, but also during manual task.

In Appendix IV Table IV.I and Table IV.II shows that there are many cells with negative values, it is due to errors in manual task, in Automatic task all values are positives. The total time depends directly on the duration of manual task. Therefore, it should be checked only automatic task, our hypothesis is true: using workflows improves all values.

The quantitative results of our study are summarized in details, each metric for the most complex activities have been evaluated. It shows that, although using workflows improves all the metrics, the errors clicks is the most improved component.

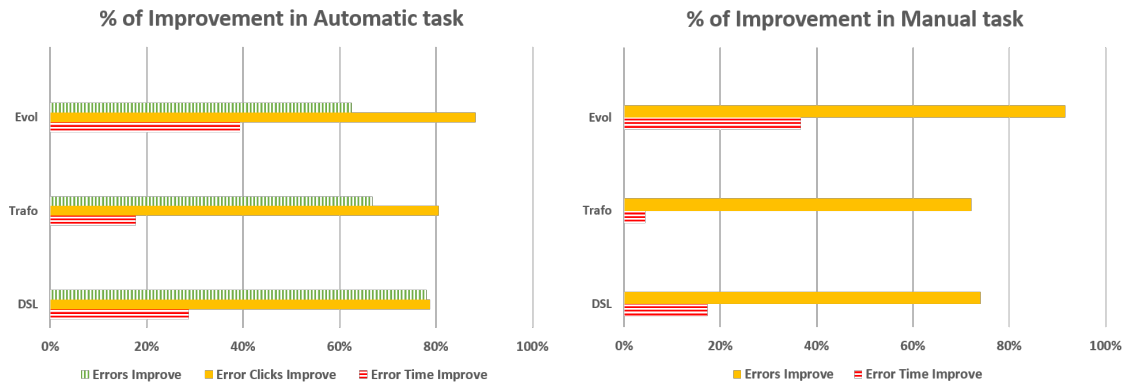
In the same way we must validate these assumptions: Does the order of method used have an influence on the dependent variables? and does the activity complexity have an influence on the dependent variables?

5.5.4 Influence of the order of the method used

H_0: There is no difference in the time T and number of clicks C to perform a task for method W than for method H.

H_1: The time and number of clicks is reduced when automating activities using method W. We need to determine if the results from group A and group B are consistent with each other: i.e., if first starting the experiment using workflows or by hand affects the results.

Figure 5.6 reports the result of a T-Test for normally distributed variables. Looking at the 2-tailed significance (in the fourth column), we observe that p-value for total time T are greater than 0.05. In contrast, if we observe in Figure 5.5 the significance (in the third column) for number of clicks C, the p-value is smaller than 0.05. Therefore, we



(a) (b)

Figure 5.7 – Percentage of Improvements in time, number of clicks and errors.

can reject the null hypothesis and we can conclude that time is affected by the order of the method used but not the number of clicks C.

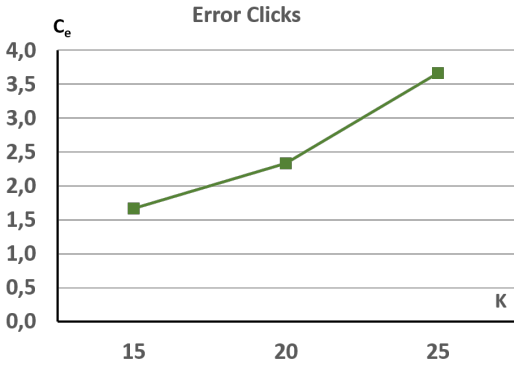
5.5.5 Influence of the activity complexity

H_0: There is no difference in the time T_e, number of clicks C_e and errors E to perform an activity with complexity $K(Evol) = 15$, $K(DSL) = 20$, $K(Trafo) = 25$.

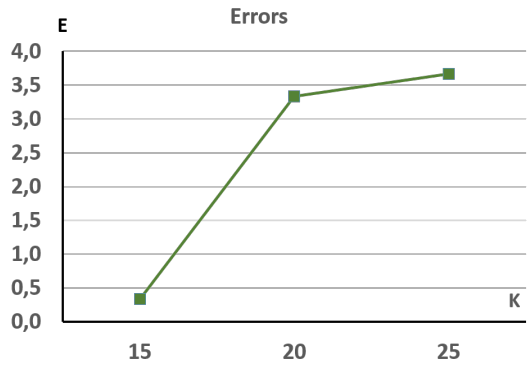
H_1: The time, number of clicks and number of errors is increased when performing an activity with complexity $K(Evol) = 15$, $K(DSL) = 20$, $K(Trafo) = 25$.

We need to determine if the results from group A and group B are consistent with each other: i.e., if first starting the experiment using workflows or by hand affects the results. The plots in Figure 5.8 reported the performances for each type of activity for group A. First, the number of clicks C_e when errors occur; second, the number of errors E. Third error Time T_e. The three plots confirm that the time, the number of clicks and errors to perform the activity increases, as the complexity of the activity increases.

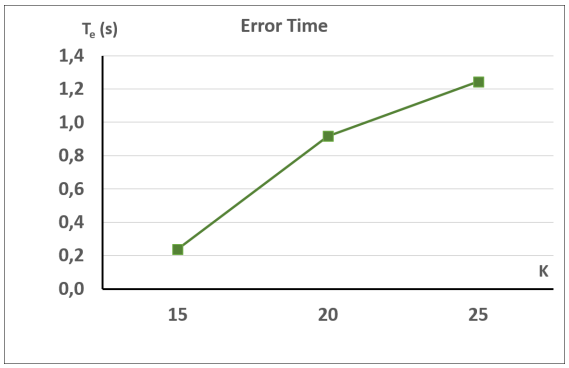
For group B Figure 5.9 shows the number of clicks C_e when errors occur; the number of errors E and finally, error Time T_e. In Figure 5.9(a) for an index of complexity $K(DSL) = 20$, the creation of DSL, the number of clicks has been greater. This is due to



(a) Number of error clicks respect to the complexity of an activity



(b) Number of errors respect to the complexity of an activity



(c) Time spent in errors respect to the complexity of an activity

Figure 5.8 – Improvements in number of clicks, errors and time with respect to the complexity of an activity in a workflow for group A.

an error in a manual activity that increases the average number of clicks to your solution. In this case this increase was due to errors in manual activities which do not affect the use of workflows. The two plots Figure 5.9(b) and Figure 5.9(c)confirm that the time, the number of clicks and errors to perform the activity increases, as the complexity of the activity increases.

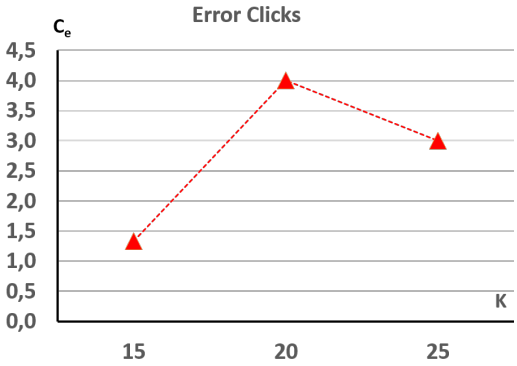
5.5.6 Significance of the results

Throughout the experiments, it was verified that the errors, time and clicks are reduced when adopting our approach versus when not. This was demonstrated statistically, but a statistically significant result is not necessarily important or meaningful [20]. To assess the substantive significance of a result we need to interpret our estimates of the effect size. To achieve this, the Cohen's d effect size will be used.

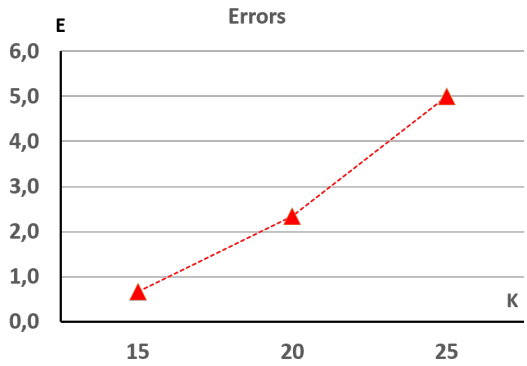
Using the Effect Size Calculator in [14] The results in Figure 5.10 were obtained. This means that improvement for T and T_e are small in effect but exist. But improvement for T_c, C, C_e and C_c has a effect very large [11]. This information highlights the importance of the obtained results. To obtain these results, the distributions of the variables mentioned were compared with the method used. That is to say, it is for example C of method W vs method H. The groups were not taken into account.

5.6 Discussion

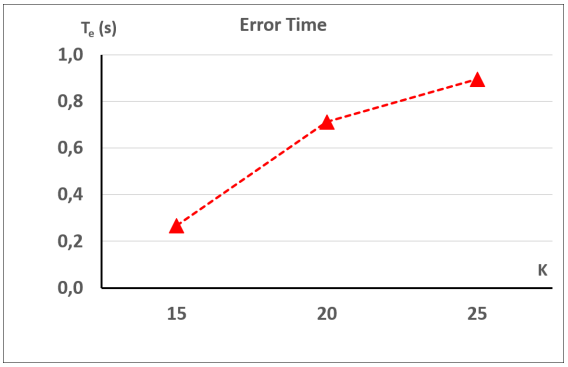
The results indicate that there is an improvement in the total time, in the number of errors, time correction of these errors and the number of clicks when errors occur. When this occurs in automated activities, it is clear that our approach significantly improves the used metrics, this means that automate activities that require less attention by the user do not only improve productivity but actually reduce the number of errors and decreases the correction time of them.



(a) Number of error clicks respect to the complexity of an activity



(b) Number of errors respect to the complexity of an activity



(c) Time spent in errors respect to the complexity of an activity

Figure 5.9 – Improvements in number of clicks, errors and time with respect to the complexity of an activity in a workflow for group B.

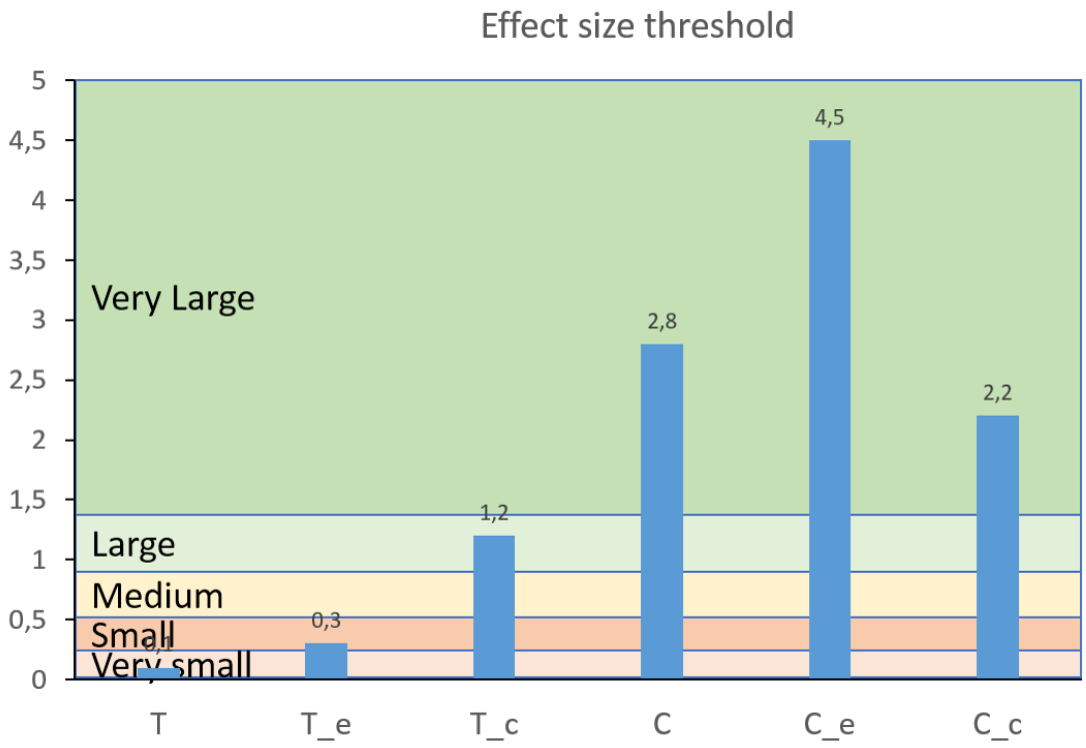


Figure 5.10 – Effect size threshold showing Cohen's D value

On the other hand, it is clear that an error in manual activities can increase the total time and error time for error correction and also this can generate new errors. Therefore, very important results shown as using workflows helps the user to focus on manual activities, essential activities to possibly reduce errors in these activities due to greater concentration of user as we can see in the figure Figure 5.7(b).

As seen in Figure 5.7(a), the variable that has a greater improvement in automated task is the number of clicks on errors C_e. It is clear that this improvement is given by the fact of facilitating the search and selection of toolbars and models when workflows is used. Fewer clicks are used when you do not have to navigate in a file structure that is not known even by experts in AToMPM. In Figure 5.7(b) the variable that has a greater improvement in manual task is the number of errors E. This is because in these activities is necessary to know the model location and its extension. Almost all participants had errors writing the file extension, so the percentage of improvement using method W is around 70% because by using workflows is not necessary to know the model extension.

By error type the most common error in automatic and manual activities is the functional error, which is when you click on the button or tab wrong. These errors are minimized when using workflows since the number of objects on which you must click are minimized. That is, fewer button and menus, represent less functional errors.

Noting the results of the survey, it is clear that 100% of users found useful to use workflows and they said that when using workflows there is an improvement and it is easier to perform modeling activities in AToMPM.

As a conclusion, among our five hypotheses, four are verified, and the answers for our research questions are:

RQ1: for the extent of the experiments conducted, the total time for mechanical and efforts of the real users is reduced when automating activities.

RQ2: Having confirmed our hypothesis about errors, the number of errors is reduced

when using workflows to automate activities for manual and automated task. All time, number of clicks, and error variables have a smaller value for method W than for method H.

RQ3: Similar to the previous answer, the time for correcting the errors is reduced when using workflows to automating activities for automated task for each type of errors.

5.7 Threats to Validity

Threats to construct validity refer to the extent to which the experimental setting actually reflects the construct under study. Participants may try to figure out what the purpose of the experiment is and to base their behavior on this. We minimized this threat by concealing the goal of the experiment to them. Unlike the first study in Chapter 4, we included the time and effort for manual tasks, which may have a negative influence on the results if they take longer than the automatic tasks.

Threats to internal validity are related to the influences that can affect the factors with respect to causality. Our evaluation had the effect that subjects react differently as time passes (because of fatigue).We solved this threat by dividing the experiment into different activities. We also preprocessed inconsistent data in order to eliminate outliers.

Threats to external validity refer to conditions that limit our ability to generalize the results of the experiment. In our experiment, the subject population may not be representative of the entire population that we want to generalize. To deal with this threat, we used a confidence interval of 95%. This means that if conclusions followed a normal distribution, the results would be true 95% of the times every time evaluation is repeated.

Threats to conclusion validity concern the relationship between the treatment and the outcome. In our experiment, subjects within a group have more experience with the tools than others. This threat was minimized with a convenience sampling when selecting

participants, since the study required experienced users of AToMPM. All participants had a comparable experience. This threat was also minimized by providing the subjects with a tutorial for workflows, which helped them to solve the problem at hand. This also minimized learning curve. Also, our experiments may be threatened by the reliability of our measures. We used time and click measures, which are objective measures that are more reliable than subjective measures. In addition, the precision of the measures may have been affected because the time for activity completion included the time to think about solutions and look for possible answers. To reduce this threat, we observed subjects while they were performing the different activities in order to guarantee their exclusive dedication to the activities and to monitor the relevant times.

CHAPTER 6

IMPROVING THE WORKFLOW LANGUAGE

6.1 Parameter Dependency

After analyzing the results of the survey from Chapter 5, several responses showed that setting run-time parameters could be improved. e.g., "if multiple steps of the workflow required the same parameters, I did not expect to introduce them more than one time"[Participant 2]. This motivated us to improve the workflow language to avoid entering the same value for multiple run-time parameters. For this purpose we introduce the concept of parameter dependency. Within the same workflow, several tasks may share the same parameters, for example, in the DSL activity, we have to enter the location parameter for the `SaveModel` and `GenerateAS`.

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.

6.1.1 New features

This generates changes to the implementation discussed in Chapter 3 which are described below. The metamodel of the workflow language in Figure 3.1 must be modified as shown in Figure 6.1. We add new class that represents dependencies between parameters. `srcParam` is the attribute to define the task from which the parameter is taken

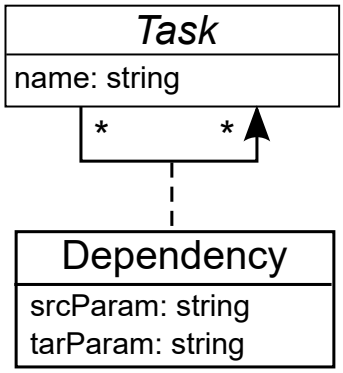


Figure 6.1 – New metamodel of workflows for modeling tools

(source). `tarParam` is the attribute to define the task that takes the parameter (target).

Figure 6.2 shows the improved workflow of the transformation example from Figure 5.2 using dependency links depicted as dashed arrows.

These dependencies assign the parameter, is this case, the location parameter from `GeneratePMM` task to `LoadToolbar` task, the first two tasks of the workflow. When the user presses the `LoadWorkflow` button, the simulation (presented in Figure 3.2) creates an instance of the parameter object and pops up a dialog prompting with the `GeneratePMMM` location but not the `LoadToolbar` location because it depends on the first task. Dependencies effectively reduce the number of run-time parameters to fill out by the user.

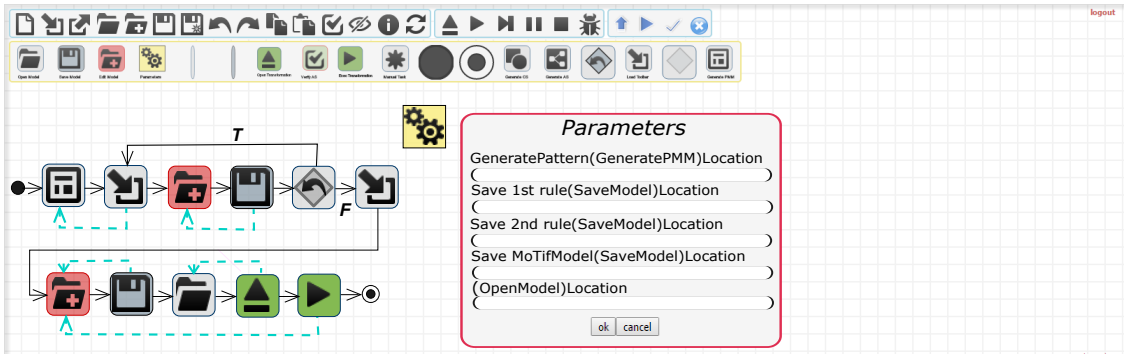


Figure 6.2 – Workflow of the transformation example with dependencies.

Similarly, Figure 6.3 shows the improved workflow to create a DSL from Figure 5.1.

6.2 Revising the Complexity of Activities

The introduction of dependencies has an impact on the activity complexity. Since dependency is optional, we need to revise equation 5.1 to be more general. This gives us a quantitative measure to compare workflows. Equation 6.1 gives the generalized formula for complexity:

$$K(Act) = K_0 + k_a \times (P(Act) - D(Act)) + k_m \times M(Act) \tag{6.1}$$

where $D(Act)$ is the number of dependency links in the workflow.

As seen in Chapter 5 we can have an optimal mechanical time associated with Fitt's Law time and time associated with the complexity of an activity. This is:

$$T_m = T_c + T_{FL} = 258 \times K(Act) + \sum_{D,S} a + b \times log_2(1 + D/S) \tag{6.2}$$

Where $K(Act)$ is the new formula to complexity. D is the distance from a given cursor position to the position of a widget to reach and S is the smallest value of the width or height of the widget. This time applies to all clicklable objects with size S and all

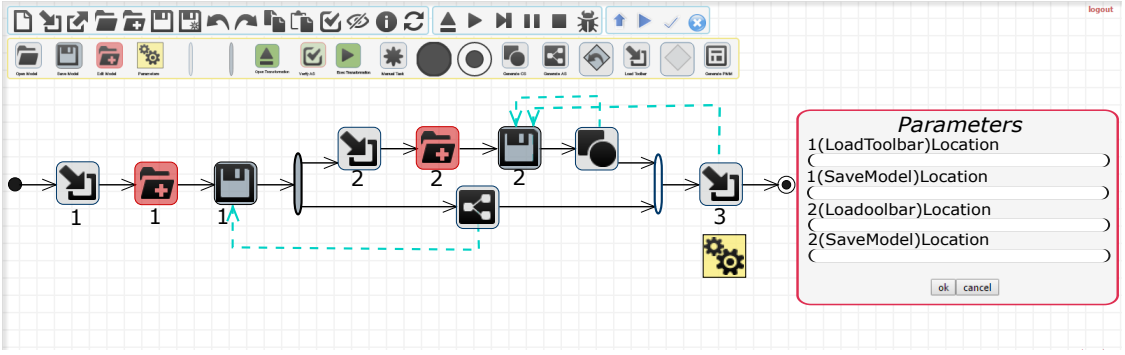


Figure 6.3 – Workflow to create a DSL using dependencies.

distances between valid objects D. By making the respective replacements, we obtain:

$$T_m = 258 \times (7 + 2 \times (P(Act) - D(Act)) + 2 \times M(Act)) + \sum_{D,S} 166.75 + 155.93 \times log_2(1 + D/S) \tag{6.3}$$

This is the expected optimal mechanical time T_m for a given activity with complexity $K(Act)$ on a 24 inch screen with a resolution of 1920×1080 pixels.

CHAPTER 7

CONCLUSION

We conclude by summarizing the contributions of this thesis outlining future work. The work presented in this thesis makes several contributions to the fields of MDE, model transformations and automation in software engineering.

7.1 Summary

We began our work by approaching the problem presented in MDE tools, which is that each tool provides many common functionalities but the process to use these functionalities differs greatly depending on the tool used. This is a complex process for the user who requires heavy mental loads that increase the accidental complexity of the tools used.

7.1.1 Design of a Reusable Workflow Language

We presented an MDE-based solution where MDE developers define workflows that can be executed. The execution of workflows is implemented as a model transformation. This provides a great advantage, since the execution specification is reusable and portable. Our approach automates common task that users perform in their every day activities when using a MDE tool, including the integration of manual tasks. As a result, our reusable workflow language is a MDE approach that improves MDE activities.

7.1.2 Analysis of the improvement when using workflows

The goal of this approach is to increase the productivity of modelers by automating the common tasks they perform in AToMPM. We performed an empirical study in or-

der to evaluate the improvement on the users' productivity. For this test. the analysis assumed perfect users, ignoring the human factor. The results show that workflows improve by 45% mechanical efforts and by 57% the cognitive efforts (in terms of time to completion) .

7.1.3 User study to evaluate the improvement in productivity when using workflows

The goal of the second study is to analyze different activities MDE, and measure the improvement in productivity with real human users, this confirming the results shown in the first evaluation. To this end, we conducted a user study to evaluate the impact of the user errors in our approach. The results give evidence that, using workflows in a real scenario, modelers improve their productivity in the MDE tool AToMPM. Error occurrence is reduced not only for automatic tasks by 60%, but also for manual tasks by 70% for each activity. The time for correcting the errors is also reduced when using workflows to automate the activities by 20% for automatic task and 15% for manual task. Likewise, the number of clicks in errors was improved for all activities by 75%.

7.1.4 Improving the Workflow Language

We improved our reusable workflow language based on the feedback received from the user study. In particular, we offer a mechanism to avoid entering the same value for multiple run-time parameters. To this end, we introduce the concept of parameter dependency. This allows us to compare activities by their complexity.

7.2 Outlook

We already incorporated some improvements to our work like the concept of dependency and the task complexity. We plan to integrate more features of AToMPM in our

prototype to allow designers to define workflows for nearly any interaction process the tool can offer. We also plan to implement this approach in other MDE frameworks, such as EMF, in order to further generalize the reusability aspect of the metamodel of workflows. We would also like to implement the approach over a framework that supports multi-level modeling, such as Metadepth, to further reduce the number of interactions with the user. Hopefully, the contribution of this thesis will help MDE developers produce software models faster, more efficiently and with fewer errors; thereby delivering higher quality software.

BIBLIOGRAPHY

[1] Advance Key (2016). Advance Key and Mouse Recorder. http://mouse-recorder.macro-expert.com/. Accessed: 2016-03-25.

[2] Alajrami, S., Romanovsky, A., Watson, P., and Roth, A. (2014). Towards Cloud-Based Software Process Modelling and Enactment. In *Model-Driven Engineering on and for the Cloud*, volume 1242 of *CloudMDE'14*, pages 6–15.

[3] Atkinson, C. (1997). Meta-modelling for distributed object environments. In *Enterprise Distributed Object Computing Workshop*, pages 90–101. IEEE.

[4] Atkinson, C. and Gerbig, R. (2012). Melanie: Multi-level Modeling and Ontology Engineering Environment. In *International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2. ACM.

[5] Atkinson, C. and Kühne, T. (2001). The Essence of Multilevel Metamodeling. In *Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 19–33. Springer.

[6] AToMPM (2013). AToMPM tutorial. http://www.slideshare.net/eugenesyriani/atompm-introductory-tutorial. Accessed: 2015-08-07.

[7] Buzan, T. (2006). *The ultimate book of mind maps: unlock your creativity, boost your memory, change your life*. HarperCollins UK.

[8] CamStudio (2016). Cam Studio. http://camstudio.org/. Accessed: 2016-03-25.

[9] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645.

[10] de Lara, J. and Guerra, E. (2010). Deep Meta-modelling with METADEPTH. In *Objects, Models, Components, Patterns*, volume 6141 of *TOOLS'10*, pages 1–20. Springer.

[11] Ellis, P. D. (2010). *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge University Press.

[12] EMFText (2014). EMFText screencast. `http://www.emftext.org/index.php/EMFText_Getting_Started_Screencast`. Accessed: 2015-08-07.

[13] Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G. D. P., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V. A., Visser, E., van der Vlist, K., Wachsmuth, G. H., and van der Woning, J. (2013). *The State of the Art in Language Workbenches*, pages 197–217. Springer International Publishing, Cham.

[14] ESC (2016). Effect Size Calculator. `http://www.polyu.edu.hk/mm/effectsizefaqs/calculator/calculator.html`. Accessed: 2016-06-23.

[15] France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society.

[16] Gonzalez Perez, C. and Henderson Sellers, B. (2008). *Metamodelling for Software Engineering*. Wiley Publishing.

[17] HB (2015). Human Benchmark. `http://www.humanbenchmark.com/tests/reactiontime/statistics`.

[18] Hollingsworth, D. and Hampshire, U. (1993). Workflow management coalition the workflow reference model. *Workflow Management Coalition*, 68:26.

[19] Jacob, F., Gray, J., Wynne, A., Liu, Y., and Baker, N. (2012). Domain-specific Languages for Composing Signature Discovery Workflows. In *Workshop on Domain-specific Modeling*, pages 61–64. ACM.

[20] Jacob Cohen (1992). Statistical Power Analysis. *Current Directions in Psychological Science*, 1(3):98–101.

[21] Johnson, R. and Woolf, B. (1996). The Type Object Pattern. In *EuroPLoP*.

[22] Kelly, S., Lyytinen, K., and Rossi, M. (1996). MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21. Springer.

[23] Kelly, S. and Tolvanen, J.-P. (2008). *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

[24] Kolovos, D. S., Paige, R. F., Polac, F. A., and Rose, L. M. (2007). Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69.

[25] Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2008). Novel features in languages of the epsilon model management platform. In *Modeling in Software Engineering*, pages 69–73. ACM.

[26] Kühne, T. (2006). Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4):369–385.

[27] Laprie, J.-C. (1985). Dependable computing and fault-tolerance. *Digest of Papers FTCS-15*, pages 2–11.

[28] Lara, J. D., Guerra, E., and Cuadrado, J. S. (2014). When and How to Use Multilevel Modelling. *ACM Transactions on Software Engineering and Methodology*, 24(12):1–46.

[29] Lecerof, A. and Paternò, F. (1998). Automatic support for usability evaluation. *IEEE Transactions on Software Engineering*, 24(10):863–888.

[30] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17 of *WISP '01*, page 1.

[31] Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G. M., Syriani, E., and Wimmer, M. (2014). Model transformation intents and their properties. *Software & systems modeling*, pages 1–38.

[32] Lucio, L., Mustafiz, S., Denil, J., Vangheluwe, H., and Jukss, M. (2013). FTG+PM: An Integrated Framework for Investigating Model Transformation Chains. In *SDL 2013: Model-Driven Dependability Engineering*, volume 7916 of *LNCS*, pages 182–202. Springer.

[33] Ludewig, J. (2003). Models in software engineering – an introduction. *Software and Systems Modeling*, 2(1):5–14.

[34] MacKenzie, I. S. (1992). Fitts' Law As a Research and Design Tool in Human-computer Interaction. *Hum.-Comput. Interact.*, 7(1):91–139.

[35] Mahmud, M., Abdullah, S., and Hosain, S. (2013). GWDL: A Graphical Workflow Definition Language for Business Workflows. In *Recent Progress in Data Engineering and Internet Technology*, volume 156 of *LNCS*, pages 205–210. Springer.

[36] Martin, D., Wutke, D., and Leymann, F. (2008). A Novel Approach to Decentralized Workflow Enactment. In *Enterprise Distributed Object Computing*, pages 127–136. IEEE.

[37] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*, 37(4):316–344.

[38] MPS (2015). JetBrains MPS. `https://www.jetbrains.com/mps/`. Accessed: 2015-08-07.

[39] Muller, P.-A., Fondement, F., Baudry, B., and Combemale, B. (2012). Modeling modeling modeling. *Software & Systems Modeling*, 11(3):347–359.

[40] OASIS (2007). *Web Services Business Process Execution Language*, 2nd edition.

[41] OMG (2008). *Software & Systems Process Engineering Metamodel specification*, 2.0 edition.

[42] OMG (2012). Information technology - Object Management Group Unified Modeling Language, Superstructure ISO/IEC 19505-2. `http://www.omg.org/spec/UML/ISO/19505-2/PDF`.

[43] PR (2015). Page Ruler PR. `http://blarg.co.uk/tools/page-ruler`. Accessed: 2015-08-25.

[44] Rivera, J. E., Ruiz Gonzalez, D., Lopez Romero, F., Bautista, J., and Vallecillo, A. (2009). Orchestrating ATL Model Transformations. In *Proceedings of MtATL*, volume 9, pages 34–46.

[45] Russell, N., van der Aalst, W., and ter Hofstede, A. (2006a). Workflow Exception Patterns. In *Advanced Information Systems Engineering*, volume 4001 of *LNCS*, pages 288–302. Springer.

[46] Russell, N., van der Aalst, W., ter Hofstede, A., and Edmond, D. (2005). Workflow Resource Patterns: Identification, Representation and Tool Support. In *Advanced Information Systems Engineering*, volume 3520 of *LNCS*, pages 216–232. Springer.

[47] Russell, N., van der Aalst, W., ter Hofstede, A., and Mulyar, N. (2006b). Workflow Control-Flow Patterns: A Revised View. Tech. report BPM-06-22, BPM Center.

[48] Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer*, 39(2):25–31.

[49] Screen Ruler (2015). Perfect Screen Ruler PSR. `http://www.styopkin.com`. Accessed: 2015-08-25.

[50] SPSS (2016). IBM SPSS. `http://www.ibm.com/analytics/us/en/technology/spss/`. Accessed: 2016-06-10.

[51] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2nd edition.

[52] Syriani, E. and Ergin, H. (2012). Operational Semantics of UML Activity Diagram: An Application in Project Management. In *RE 2012 Workshops*, pages 1–8. IEEE.

[53] Syriani, E., Kienzle, J., and Vangheluwe, H. (2010). Exceptional Transformations. In *Theory and Practice of Model Transformation*, volume 6142 of *LNCS*, pages 199–214. Springer.

[54] Syriani, E. and Vangheluwe, H. (2008). Programmed Graph Rewriting with Time for Simulation-Based Design. In *Theory and Practice of Model Transformation*, volume 5063 of *LNCS*, pages 91–106. Springer.

[55] Syriani, E. and Vangheluwe, H. (2011). A Modular Timed Model Transformation Language. *Journal on Software and Systems Modeling*, 12(2):387–414.

[56] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. (2013). AToMPM: A Web-based Modeling Environment. In *Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, volume 1115 of *MODELS'13*, pages 21–25. CEUR-WS.org.

[57] Van Mierlo, S., Barroca, B., Vangheluwe, H., Syriani, E., and Kühne, T. (2014). Multi-Level Modelling in the Modelverse. In *Workshop on Multi-Level Modelling*, volume 1286 of *MULTI '14*, pages 83–92. CEUR-WS.org.

[58] Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85.

[59] WMC (1999). Terminology and glossary. Technical Report WFMC-TC-1011, Workflow Management Coalition.

[60] WMC (2005). Process Definition Interface – XML Process Definition Language 2.00. Technical Report WFMC-TC-1025, Workflow Management Coalition.

Appendix I

Tutorial on Using Workflows in AToMPM

Tutorial using workflows in AToMPM

Using a workflow to modify an existing model

Introduction

The Workflow is MDE-based solution where the user defines a “workflow” that can be parametrized at run-time and executed. This Workflow is a DSL for defining activities that can be performed in MDE tools.

An activity is composed of tasks, to define concrete actions to be performed. we want to automate user’s 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 a task that can be automated, since the location of that formalism is known.

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. An activity conforming to the metamodel starts from the initial node and terminates at the final node. Tasks can be sequenced one after the other.

In this tutorial, you will learn how to use Workflows in AToMPM in order to:

- Set the run-time parameters.

- Execute the workflow.

By following these steps, you will be able to

- Open a simple model.

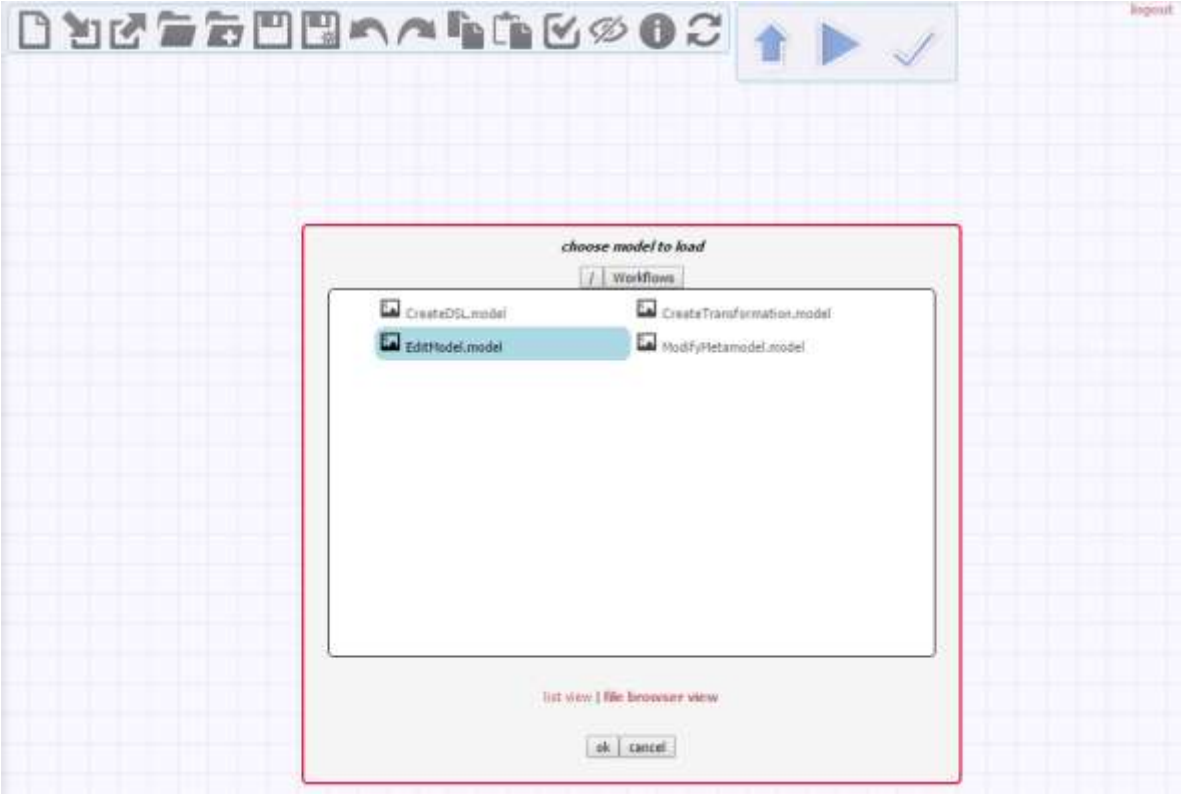
- Modify this model manually.

- Save this model.

Open the workflow

1. Click on Load Model

2. Navigate to /Workflows/EditModel.model



3. Click OK

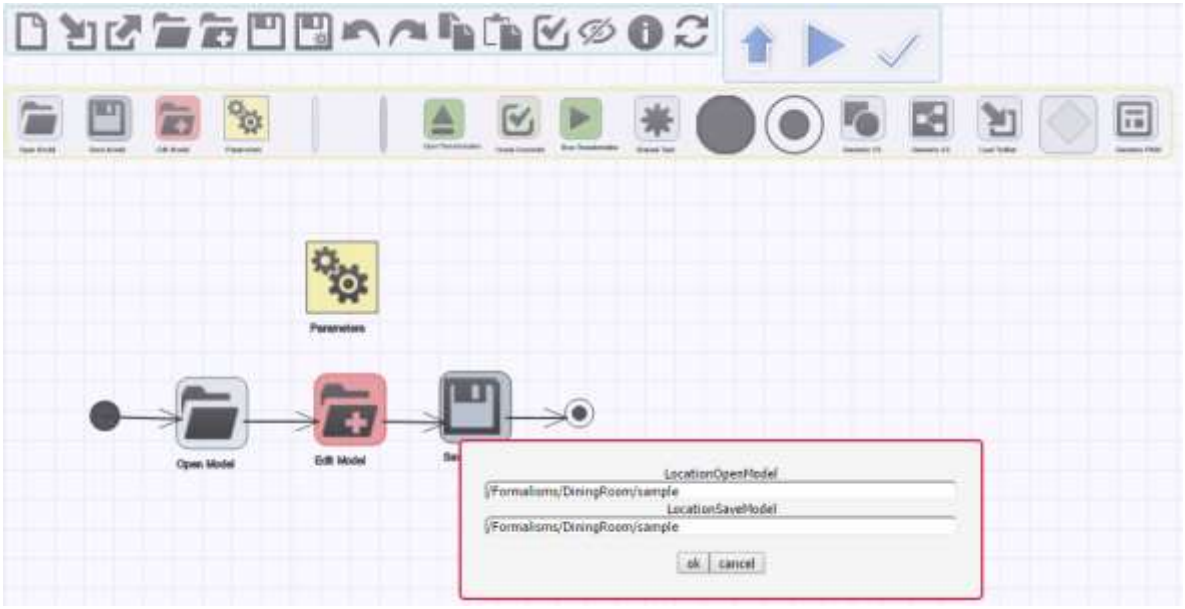
Set the run-time parameters

4. Click on the Load Parameters button

5. This pops up a dialog prompting for all required parameters. Type in these parameters

    a. Location to open the model from: /Formalisms/DiningRoom/sample

    b. Location to save the model to: /Formalisms/DiningRoom/sample

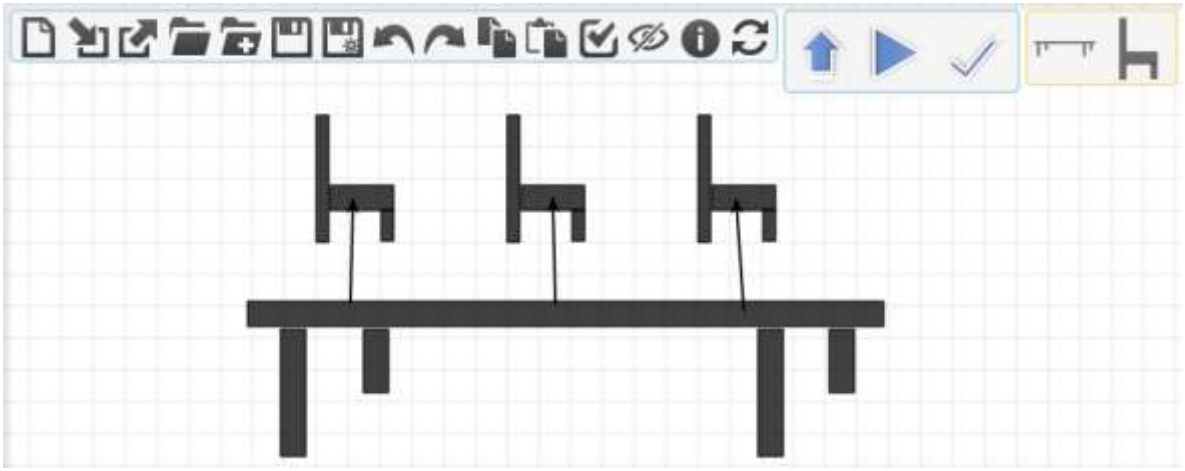


6. Click OK

Execute the workflow

7. Click on the Resume Process button

a. When a manual task is reached, a new window is opened with all necessary toolbars pre-loaded.



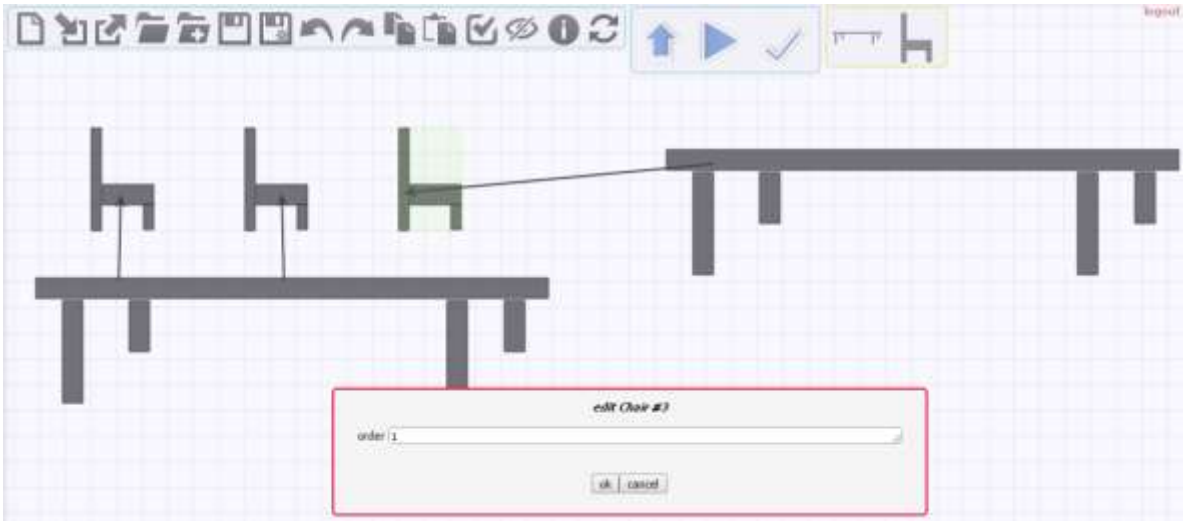
8. Modify the model manually by performing the following actions:

a. Remove the link between the table and the chair with order 3.

b. Create a new table.

c. Create a link from the new table to that chair.

d. Change the order of the chair to 1.



9. After the manual task is completed, push the Complete Task button. Then, the window closes and the simulation restarts

10. The window closes and brings you back to the window with the workflow. Push the Resume Process button

11. The model has been saved and the execution terminates.

Appendix II

User Study Directives

AToMPM Experiment

Environment requirements

Tools Description

1. CamStudio is able to record all screen and audio activity on your computer and create industry-standard AVI video files.

System Requirements

- Windows XP / Vista / 7 / 8 / 10

2. Advance Key and Mouse Recorder records your mouse actions, keyboard input and program window changes.

System Requirements

- Windows XP / Vista / 7 / 8 / 10

Tools Installation

1. To install CamStudio Software run the file camstudio.exe. To do this click Start on the task bar, select Run and then click Browse to locate the file DEBUTSETUP.EXE. The file should be located in your download or attachments directory.

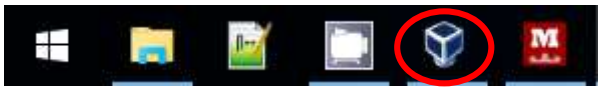
2. To Advance Key and Mouse Recorder run the file gml.exe.

3. In both cases, after the setup program has run, you will be able to use Debut or Advance Key and Mouse Recorder immediately.

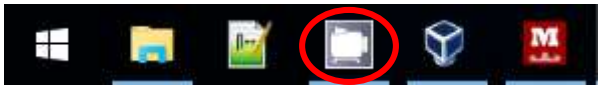
4. You can run CamStudio or Advance Key and Mouse Recorder at any time by simply clicking on the shortcut on your desktop.

Start video capture

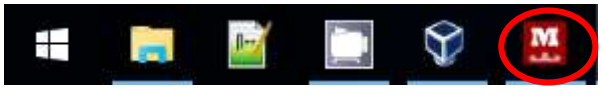
1. Open AToMPM (in Virtualbox)



2. Open CamStudio Software.

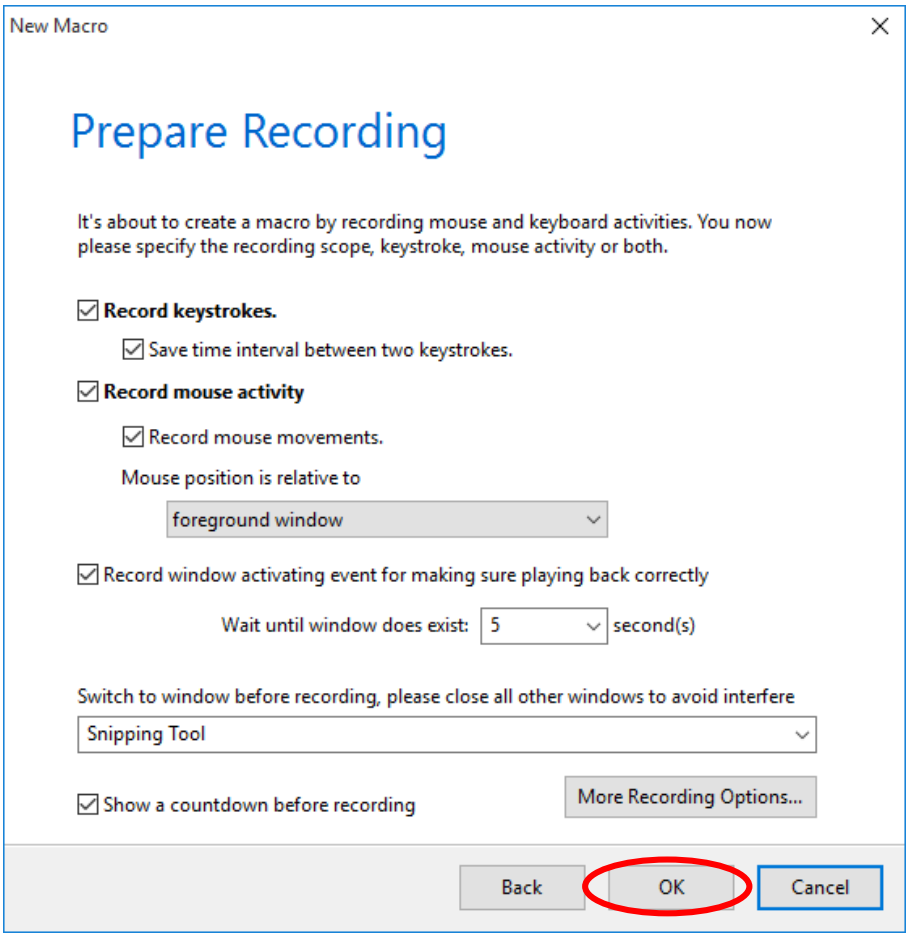


3. Open Advance Key and Mouse Recorder.



4. In Camstudio Software, click on the record button.

5. In Advance Key and Mouse Recorder, click on the ok button.



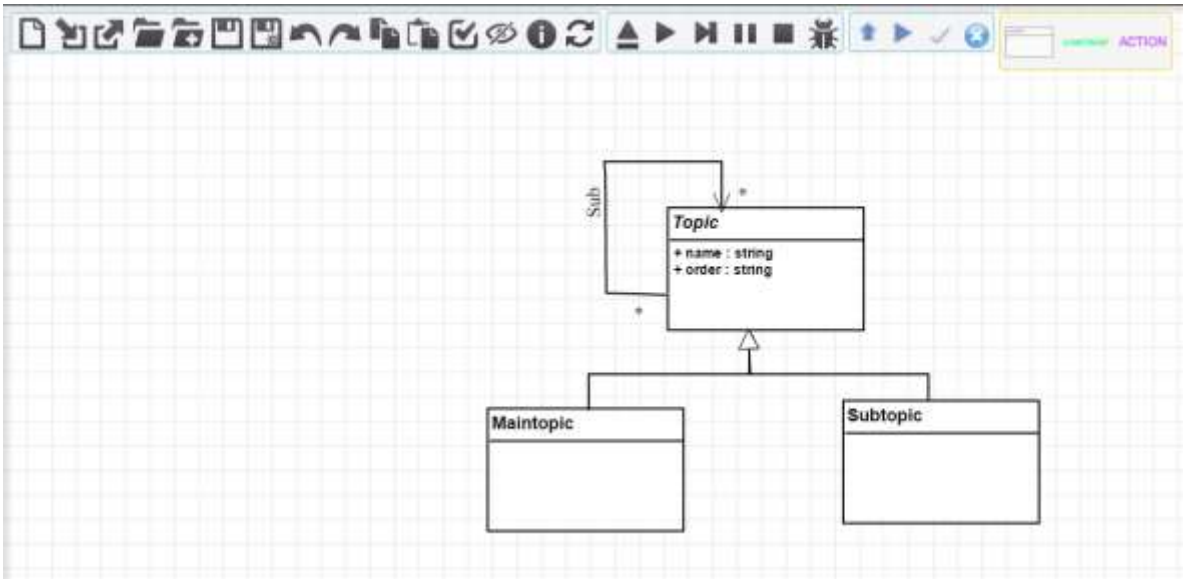
Process in AToMPM

You must perform each of the following activities in this order.

Create DSL

In this activity, you will create a simple DSL for mind maps.

1. Create the following metamodel:

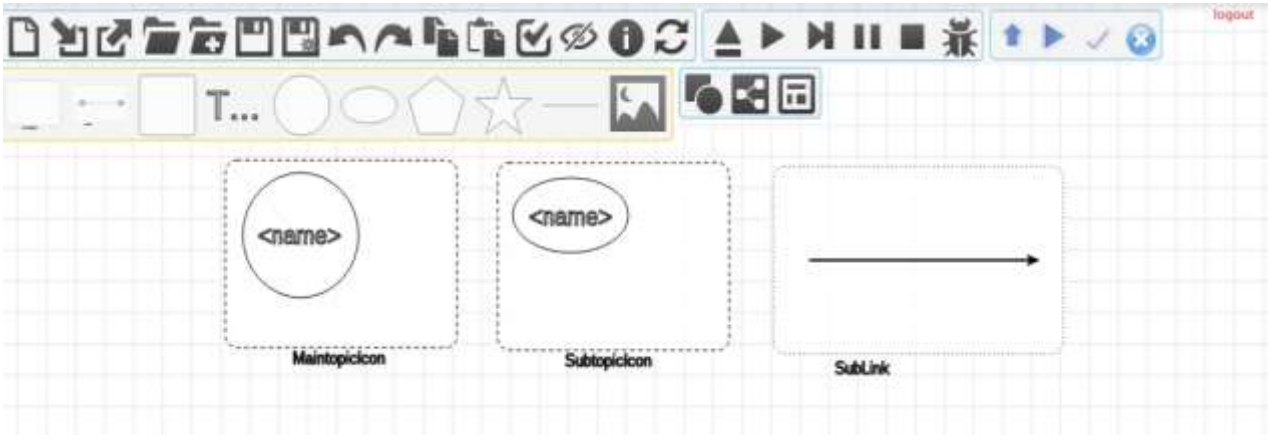


2. Save this metamodel as `mindmapMM` under `/Formalisms/MindMap/`.

3. Generate the abstract syntax from this metamodel.

4. Open a new window

5. Create the following concrete syntax model, with the following line in the mapper of both name tags: `({textContent: getAttr("name")})`



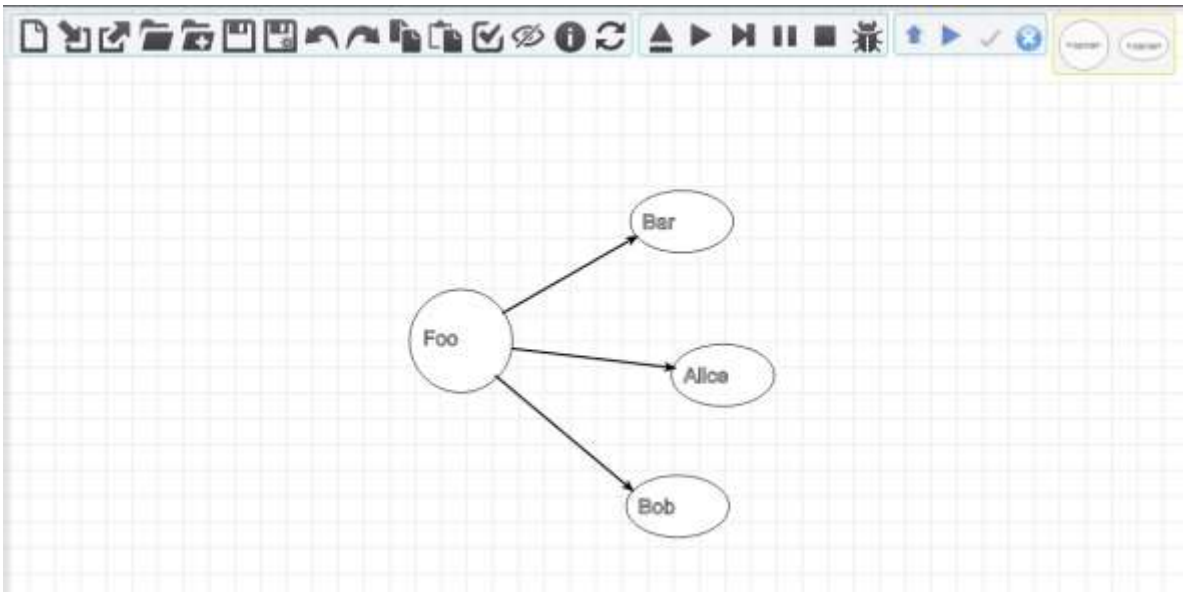
6. Save this concrete syntax model as `mindmap.simple` under `/Formalisms/MindMap/`.

7. Generate a modeling environment from this concrete syntax model.

8. Open a new window

9. Load the toolbar `mindmap.simple` under `/Formalisms/MindMap/`.

10. Create the following model:



11. Save this model as `foobar` under `/Formalisms/MindMap/`.

Create a model transformation

In this activity, you will create a model transformation that assigns orders to subtopics of a mind map.

1. In a new window, generate a modeling environment for rule patterns from the `mindmap` metamodel.

2. Create a first rule that assigns a number to each subtopic

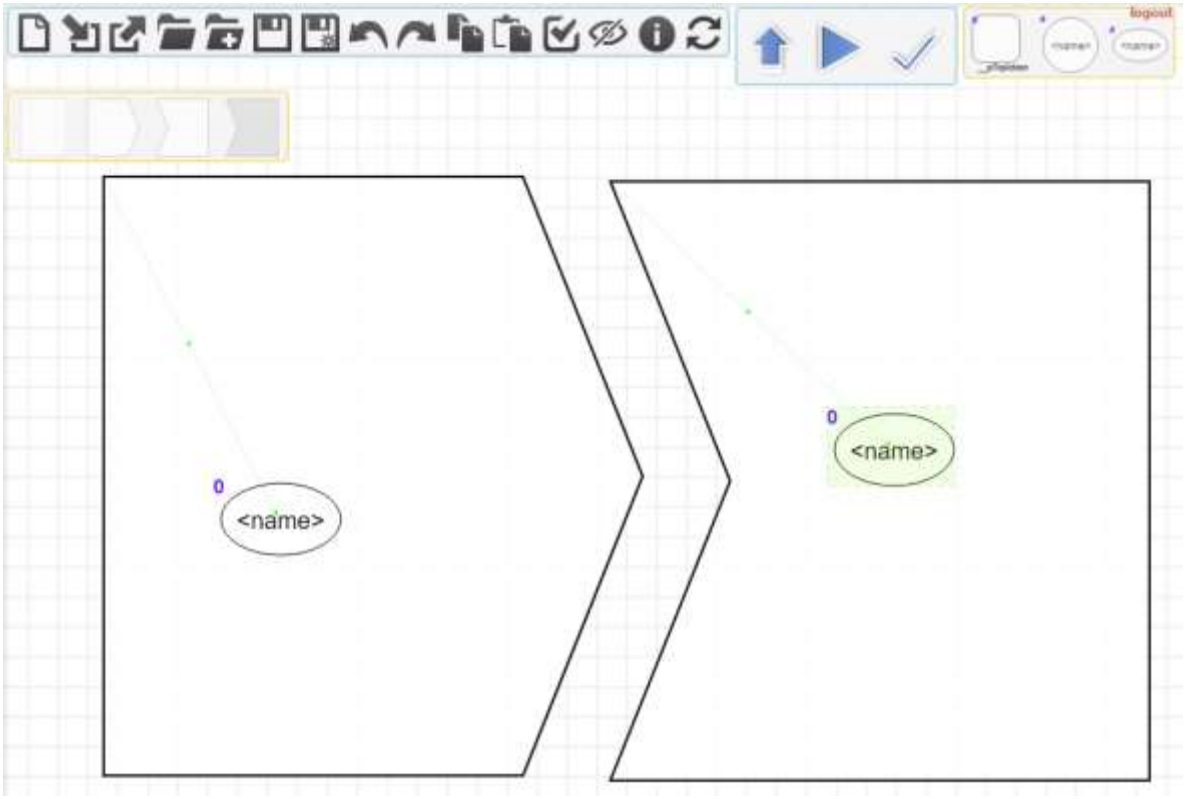
    a. The LHS condition is: `result = True`

    b. The name attribute value of the subtopic in the LHS is: `result = True`

    c. The order attribute value of the subtopic in the LHS is: `result = (getAttr() == '')`

    d. The name attribute value of the subtopic in the RHS is: `result = getAttr()`

    e. The order attribute value of the subtopic in the RHS is: `result = 1`



3. Save this rule model as `R_Assign` under `/Formalisms/MindMap/OrderSubtopics`.

4. Open a new window

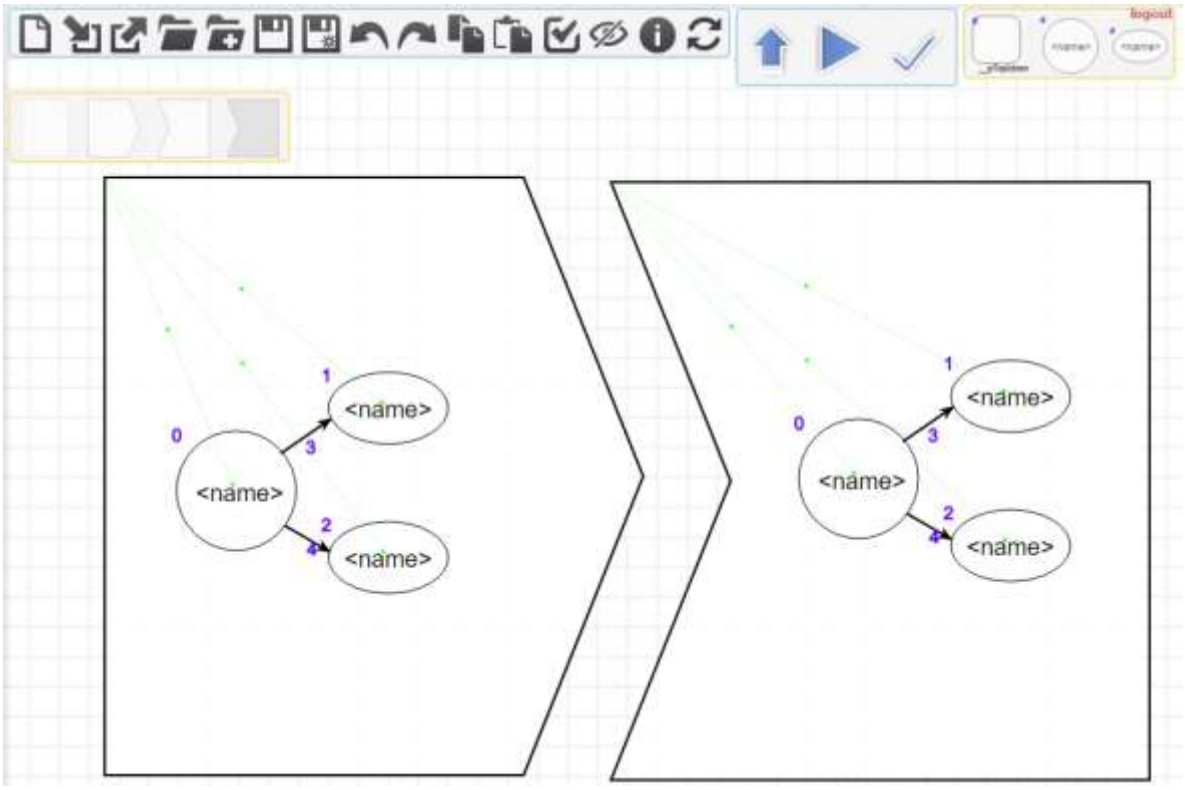
5. Create a second rule that orders subtopic uniquely

    a. The LHS condition is: `result = (str(getAttr("order", "1")) == str(getAttr("order", "2")))`

b. All name and order attribute values of the main topic and subtopics in the LHS are: `result = True`

c. All name and order attribute values of the main topic and subtopics in the RHS are: `result = getAttr()`

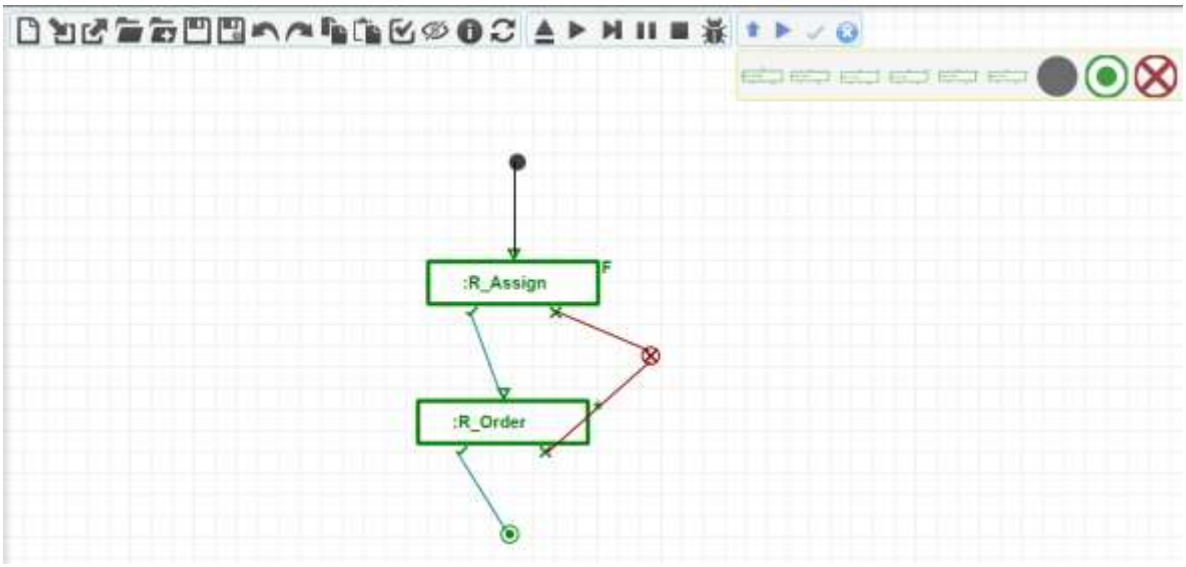
d. The RHS action is: `result = setAttr("order", getAttr("order", 1) + 1, 2)`



6. Save this rule model as `R_Order` under `/Formalisms/MindMap/OrderSubtopics`.

7. Open a new window

8. Create a MoTif model to schedule the execution of the two rules as follows:



9. Save this MoTif model as `T_OrderSubtopics` under `/Formalisms/MindMap/OrderSubtopics`.

10. Open a new window

11. Load the `foobar` model under `/Formalisms/MindMap/` that you created previously.

12. Load the transformation `T_OrderSubtopics` under `/Formalisms/MindMap/OrderSubtopics`.

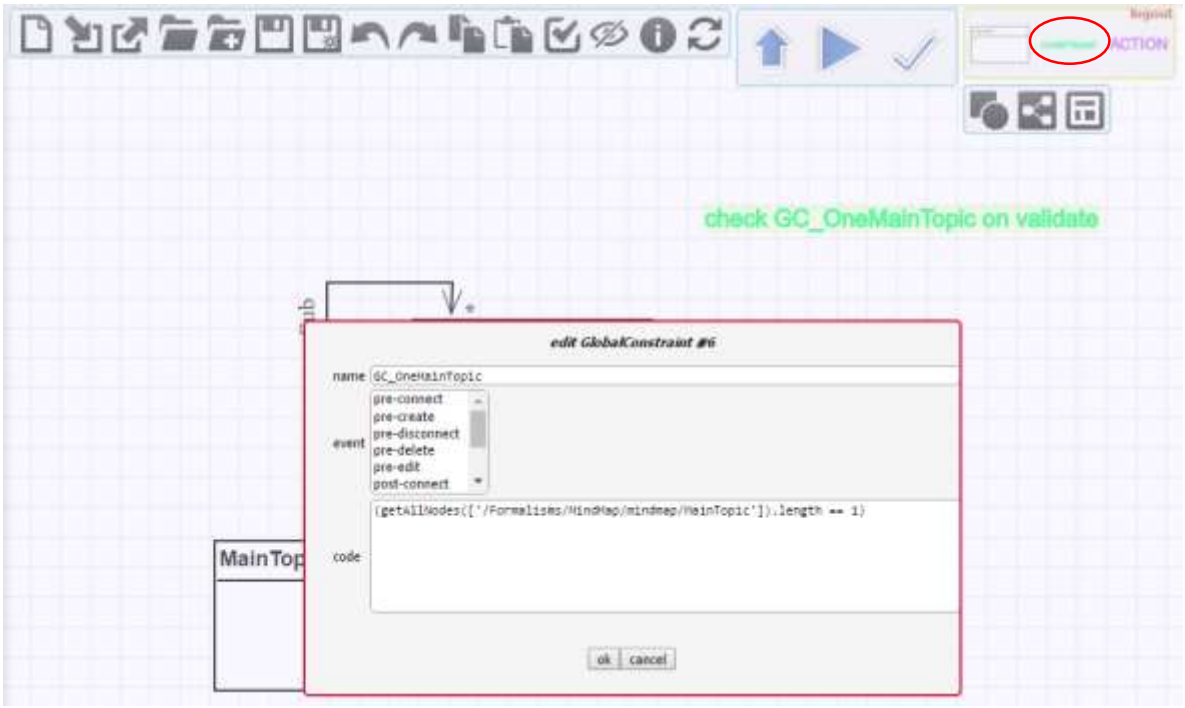
13. Wait for a few seconds and run the transformation.

Modify a metamodel

In this activity, you will modify the metamodel of mind maps by adding a constraint that we verify on the model.

1. In a new window, open the `mindmap` metamodel under `/Formalisms/MindMap/` that you created previously.

2. Add a constraint, by clicking on the circled button, called `GC_OneMaintopic` that ensures exactly one main topic is present as follows: `(getAllNodes(['/Formalisms/MindMap/mindmap/Maintopic']).length == 1)`



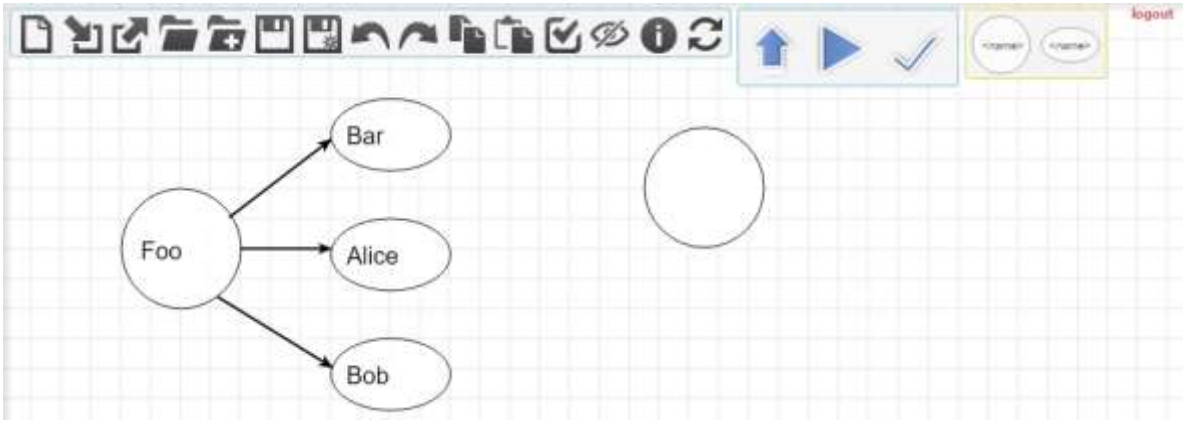
3. Save the metamodel with its current name and location.

4. Generate the abstract syntax from this metamodel.

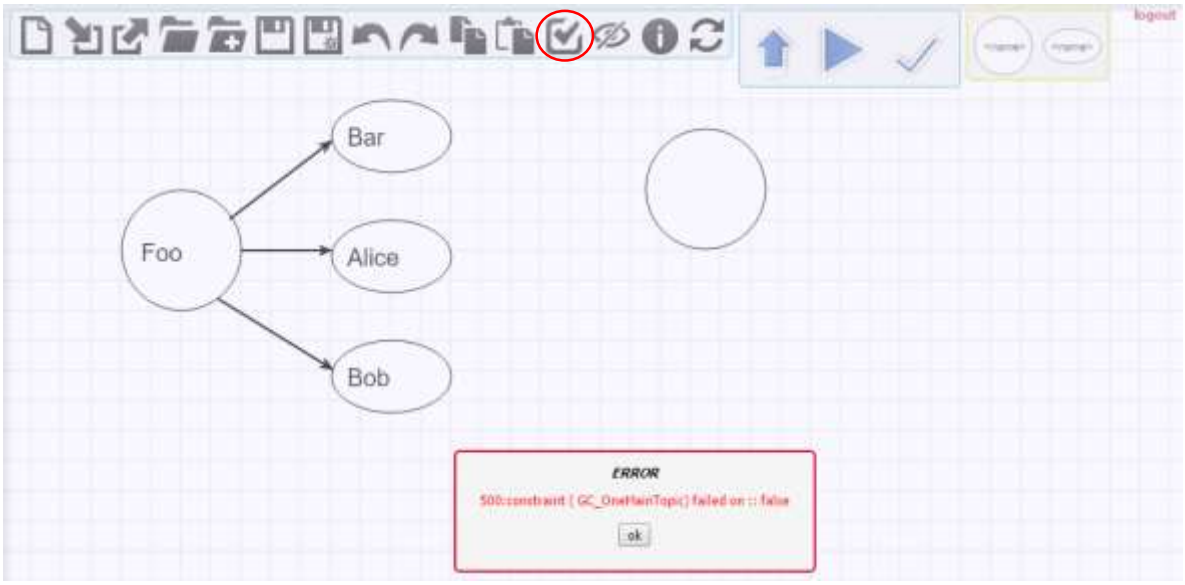
5. Open a new window

6. Load the model `foobar` under `/Formalisms/MindMap/` that you created previously.

7. Add another central topic in the model.



8. Verify the constraint by clicking on the circled button:



Paths of the toolbars

DSL

/Formalisms/__LanguageSyntax__/SimpleClassDiagram/SimpleClassDiagram

/Formalisms/__LanguageSyntax__/ConcreteSyntax/ConcreteSyntax

/Formalisms/MindMap/

Transformation

/Formalisms/__Transformations__/TransformationRule/TransformationRule
/Formalisms/__Transformations__/Transformation/MoTif

OpenModel

/Formalisms/DiningRoom/sample

/Formalisms/DiningRoom/sample

Appendix III

PostSurvey

AToMPM Experience using Workflows

1. About Workflow

Below are some questions that allow us to know your perception about the experience in AToMPM using Workflows.

1. In a scale from 1 to 5 (1 extremely easy, 5 extremely difficult) How would you rate the usage of Workflows?

1 Extremely easy	2	3	4	5 Extremely difficult
○	○	○	○	○

2. What is the most confusing part when using Workflows?

- ○ Load the parameters
- ○ Execute the activities
- ○ Complete the activities

Other (please specify)

3. During your experience with workflow, what did not work as expected?

4. When comparing both methods the old one (without Workflows) and the new one (with Workflows), which one do you prefer?

- ○ The old one (without Workflows)
- ○ The new one (with Workflows)

5. What is your favorite feature of Workflows?

- ○ The reduction of steps to accomplish a task.
- ○ The facility for accomplishing a task.
- ○ The facility for correcting mistakes
- ○ Other (please specify)

1 / 2 50%

AToMPM Experience using Workflows

2. About Tutorials

6. Were you able to find the information you were looking for on the tutorial?

- ◯ Yes
- ◯ No

7. Did you find that information helpful?

- ◯ Yes
- ◯ No

2 / 2 100%

Appendix IV

Summary of the Results of the User Study

Tables IV.I, IV.II and V.I illustrate the summary of the results obtained from the user study presented inChapter 5. For each variable on the left, the first row represents the average value obtained among the participants in a group and the second row shows the improvement when using workflows. A negative improvement indicates that by hand performed better. Time variables are shown in minutes.

	Group A					
	DSL		**Trafo**		**Evol**	
	H	*W*	*H*	*W*	*H*	*W*
T	5.40	3.03	5.52	3.07	1.67	1.38
	44%		44%		17%	
C	72	22	90	29	39	17
	69%		68%		58%	
E	13.67	3.33	12.33	3.67	1.33	0.33
	76%		70%		75%	
T_e	1.37	0.92	1.49	1.24	0.46	0.24
	30%		17%		48%	
C_e	16	2.33	18.33	3.67	13.33	1.67
	85%		80%		88%	
E_{et}	3.67	1	3	1.67	0.67	–
	73%		44%		100%	
T_{et}	0.41	0.35	0.46	0.18	0.10	–
	13%		61%		100%	
E_{ef}	4.33	1	4.67	1	–	–
	77%		79%		–	
T_{ef}	0.26	0.08	0.30	0.04	0.26	–
	69%		87%		100%	
E_{ep}	2	1.33	3	1	0.67	–
	33%		67%		100%	
T_{ep}	0.45	0.41	0.50	0.47	0.11	–
	8%		7%		100%	
E_{eb}	3.67	–	–	–	–	–
	100%		–		–	
T_{eb}	0.25	–	0.28	–	–	–
	100%		100%		–	

Table IV.I – Results for automatic tasks group A

	Group B					
	DSL		**Trafo**		**Evol**	
	H	*W*	*H*	*W*	*H*	*W*
T	4.77	2.77	5.05	2.98	2.07	2.06
	42%		41%		1%	
C	70	24	88	28	38	16
	66%		68%		57%	
E	12	2.33	13.67	5	1.33	0.67
	81%		63%		50%	
T_e	0.98	0.71	1.10	0.89	0.38	0.27
	27%		19%		30%	
C_e	14.33	4	16	3	12	1.33
	72%		81%		89%	
E_{et}	3.67	1.67	4	2.67	1	0.33
	55%		33%		67%	
T_{et}	0.36	0.35	0.34	0.28	0.22	0.02
	4%		19%		92%	
E_{ef}	3.33	0.33	4.67	2	–	–
	90%		57%		–	
T_{ef}	0.25	0.19	0.22	0.18	0.01	–
	24%		19%		100%	
E_{ep}	2.67	0.33	2.33	0.33	0.33	0.33
	88%		86%		–	
T_{ep}	0.29	–	0.33	0.33	0.15	–
	100%		1%		100%	
E_{eb}	2.33	–	2.67	–	–	–
	100%		100%		–	
T_{eb}	0.19	–	0.21	–	–	–
	100%		100%		–	

Table IV.II – Results for automatic tasks group B

Appendix V

Summary results Manual Task

	Group A						Group B					
	DSL		**Trafo**		**Evol**		**DSL**		**Trafo**		**Evol**	
	H	*W*	*H*	*W*	*H*	*W*	*H*	*W*	*H*	*W*	*H*	*W*
T	18.03	12.17	17.59	14.32	2.7	2.11	8.95	12.54	9.71	23.23	0.38	1.56
	33%		19%		22%		-40%		-139%		-312%	
E	11	2.67	11	2.67	1.67	–	9.67	2.67	12.67	4	2	0.33
	76%		76%		10%		72%		68%		83%	
T_e	3.93	2.74	4.47	3.73	1.39	0.71	2.56	2.45	2.64	2.84	1.15	0.87
	30%		17%		49%		4%		-8%		25%	
E_{et}	3.33	1.33	2.67	0.67	0.33	–	3.67	2	3.33	1.67	1	0.33
	60%		75%		10%		45%		50%		67%	
T_{et}	1.45	1.1	1.39	0.54	0.29	–	0.79	1.83	0.82	0.88	0.66	0.04
	24%		61%		10%		-130%		-8%		93%	
E_{ef}	2.67	0.33	3.67	0.67	0.67	–	3	0.67	4.33	1.33	0.33	–
	88%		82%		10%		78%		69%		10%	
T_{ef}	0.84	0.24	0.89	0.12	0.77	–	0.51	0.62	0.53	0.57	0.04	–
	72%		86%		10%		-22%		-8%		10%	
E_{ep}	2.33	1	3	1.33	0.67	–	1.33	–	3	1	0.67	–
	57%		56%		100%		100%		67%		100%	
T_{ep}	0.84	1.4	1.34	3.07	0.33	–	0.77	–	0.79	1.39	0.45	–
	-68%		-129%		100%		100%		-76%		100%	
E_{eb}	2.67	–	1.67	–	–		1.67	–	2	–	–	
	10%		100%				100%		100%			
T_{eb}	0.8	–	0.85	–	–		0.49	–	0.5	–	–	
	100%		100%				100%		100%			

Table V.I – Results for manual tasks

Appendix VI

Step to create a DSL in AToMPM and EMFText

The following are the steps to create a DSL in ATOMPM and later in EMFText:

1. Define the abstract syntax. Figure VI.1 to Figure VI.6 depicts how the user has to load severals toolbars, for example, the class diagram formalism that is a toolbar to create entities that conform the classes in a class diagram. Graphically by using this toolbars the user can build a model, in this case a metamodel. Then, he generates the abstract syntax of the DSL from that metamodel by loading the compiler toolbar. He then generates the domain-specific modeling environment by saving and compiling this metamodel.
2. Define the concrete syntax. Figure VI.7 to Figure VI.13 shows the process to create a concrete syntax. For that, the user has to load the concrete syntax formalism that is a toolbar to create icons that represent our DSL and assign a concrete syntax e.g., icon to each individual class and association from the metamodel by drawing lines and shapes.
3. Build a Model. Finally, by using the concrete syntax created, the user can define a new model. Figure VI.14 to Figure VI.16 shows how the user build a model by loading a toolbar (the new DSL created in precedent steps) and drawing in the canvas.

In contrast, the steps are different to create a DSL in EMFText [12]. The figures in Figure VI.17 to Figure VI.31 depicts the process. For resume, the language designer first creates a new project by specifying the project settings in the wizard dialog. He then creates an file (Ecore diagram extension) and graphically builds the metamodel. Next, he needs to create a model (Specifically a generator model that is model to generate the DSL environment) from the metamodel file. To define the concrete syntax, he creates a

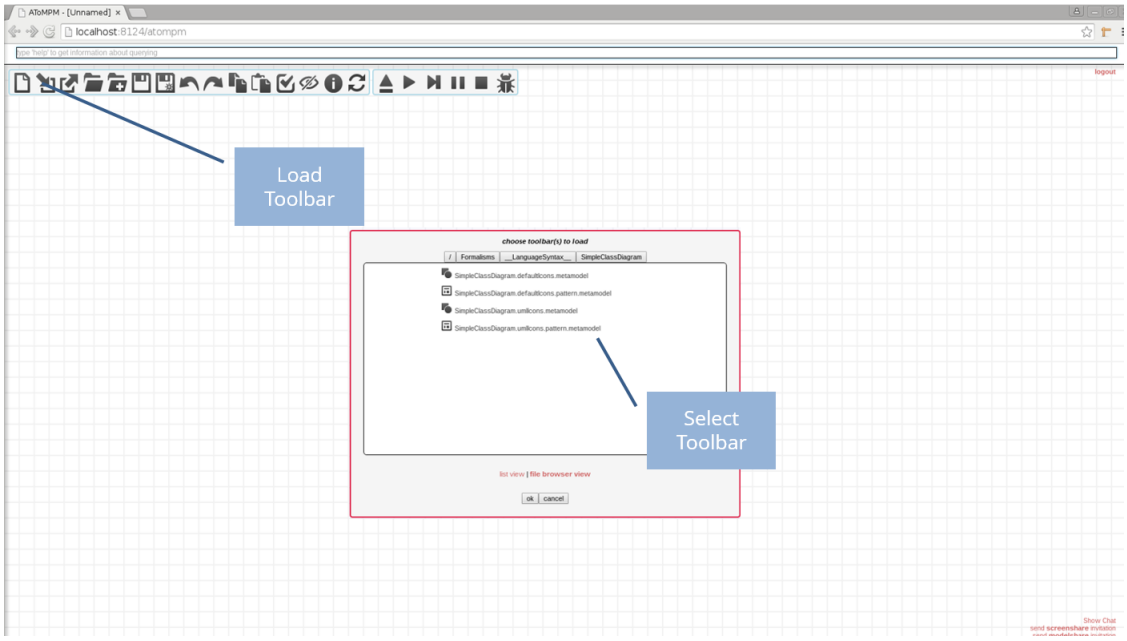


Figure VI.1 – Step 1 to create a DSL in AToMPM

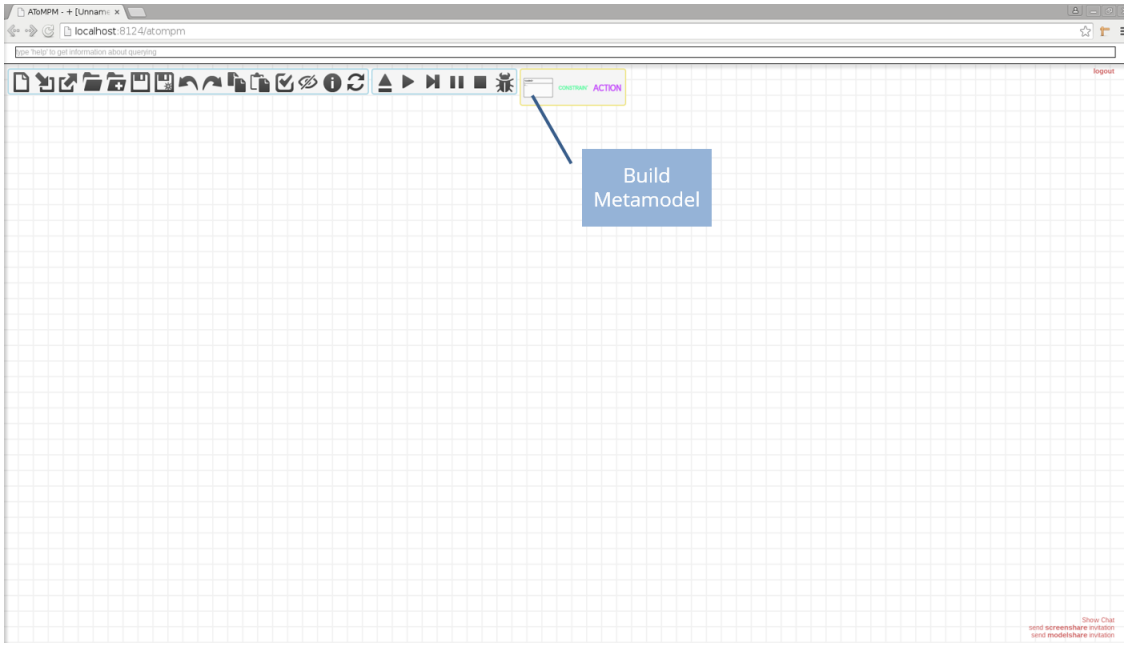


Figure VI.2 – Step 2 to create a DSL in AToMPM

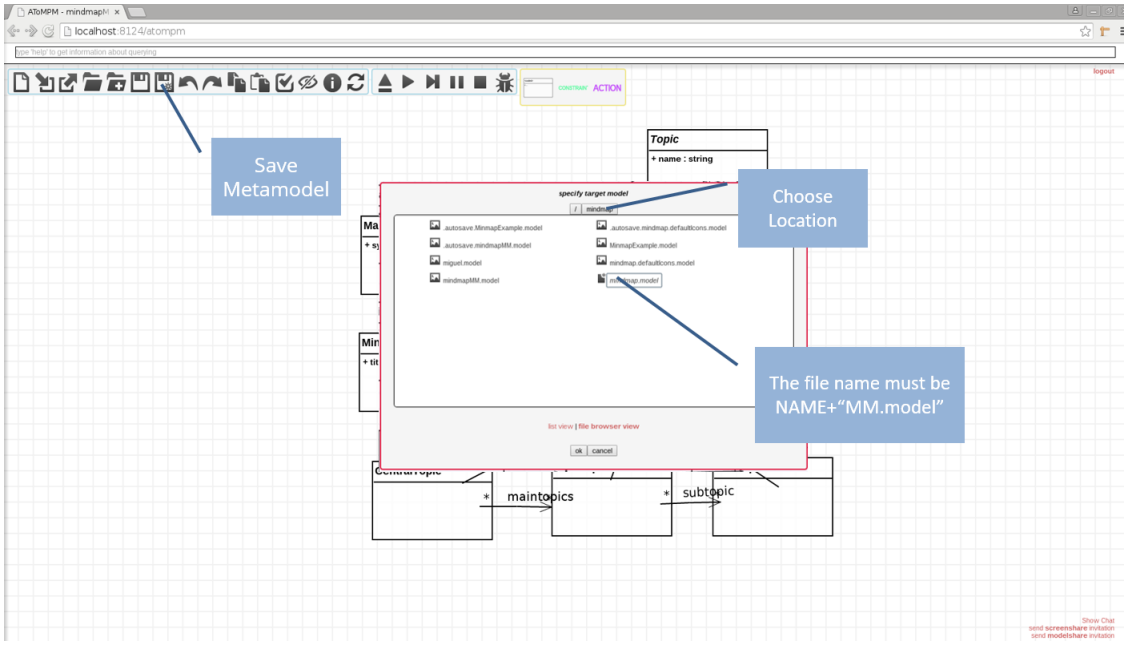


Figure VI.3 – Step 3 to create a DSL in AToMPM

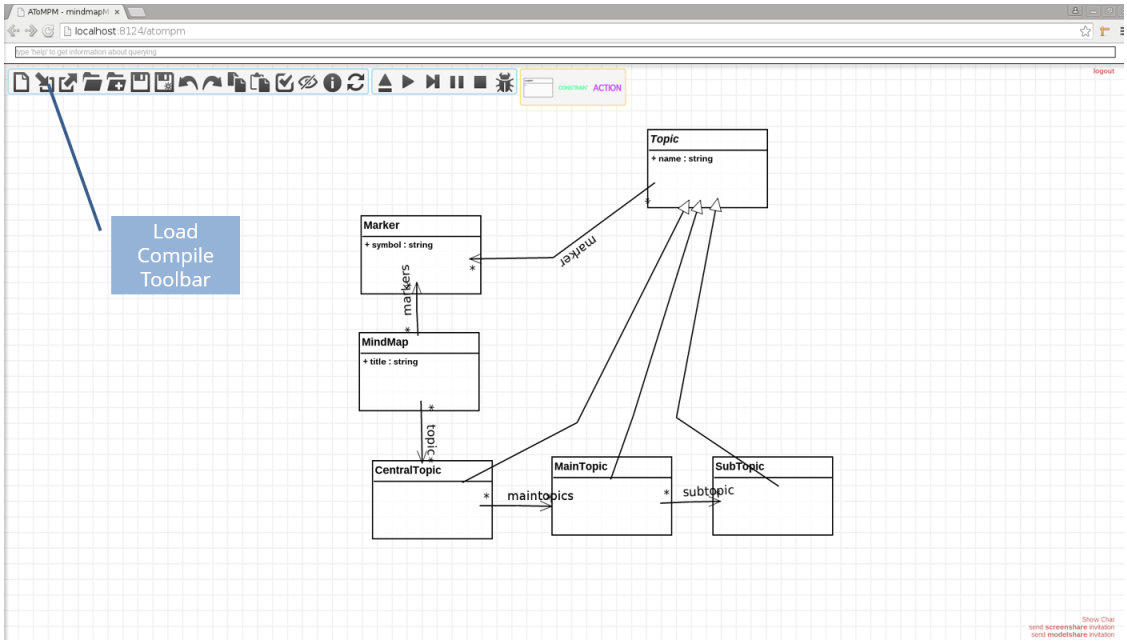


Figure VI.4 – Step 4 to create a DSL in AToMPM

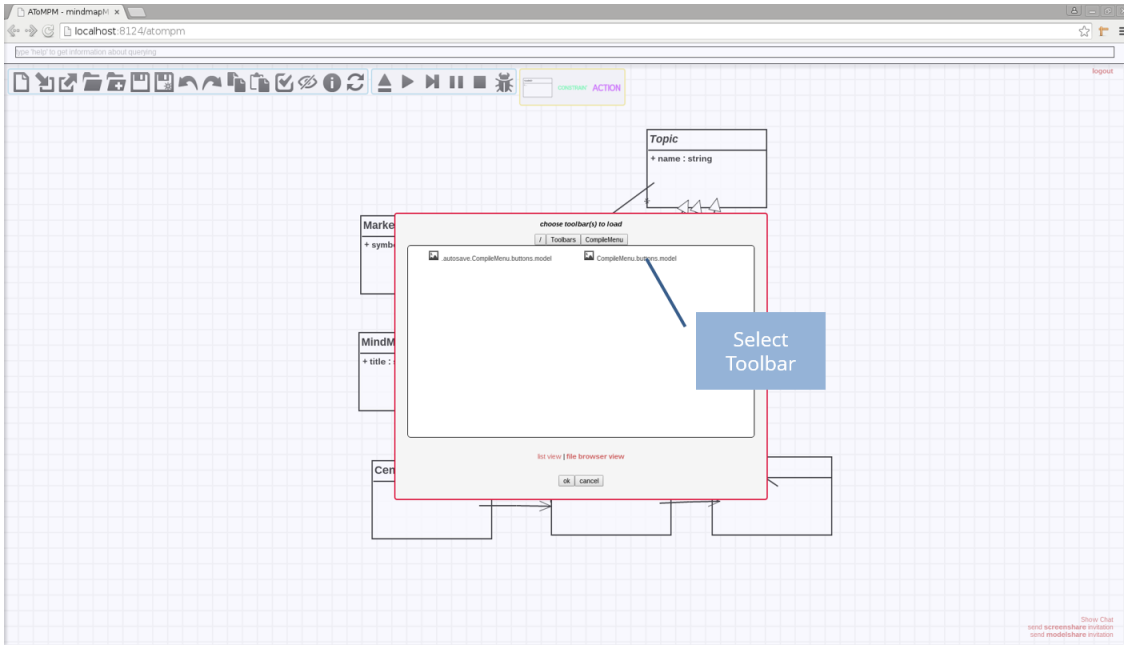


Figure VI.5 – Step 5 to create a DSL in AToMPM

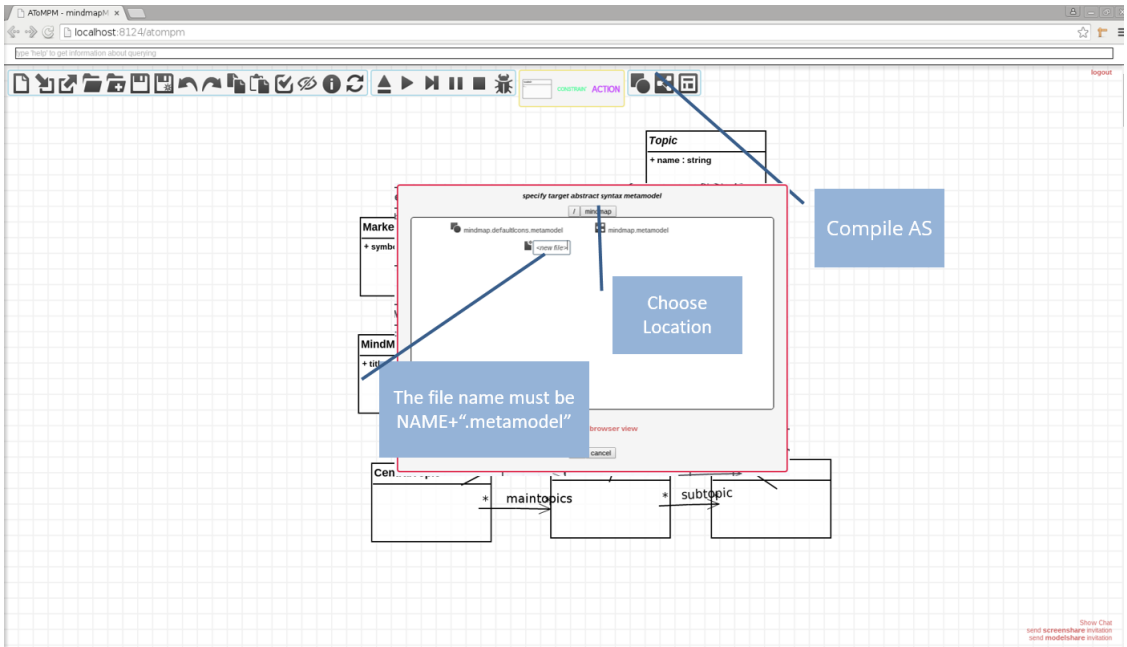


Figure VI.6 – Step 6 to create a DSL in AToMPM

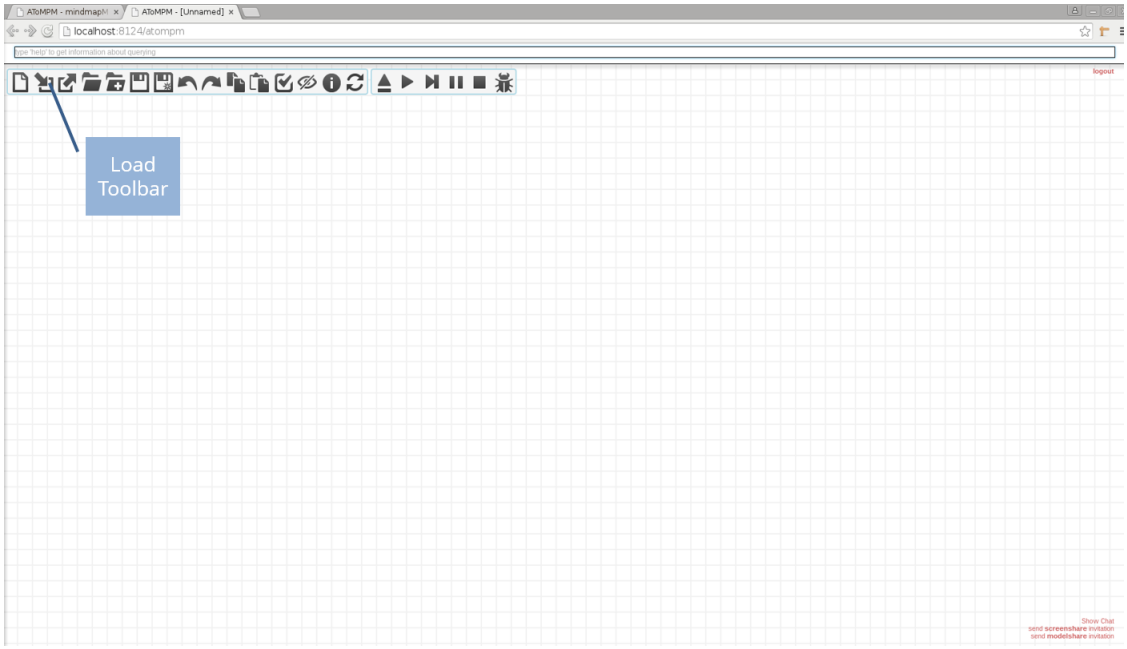


Figure VI.7 – Step 8 to create a DSL in AToMPM

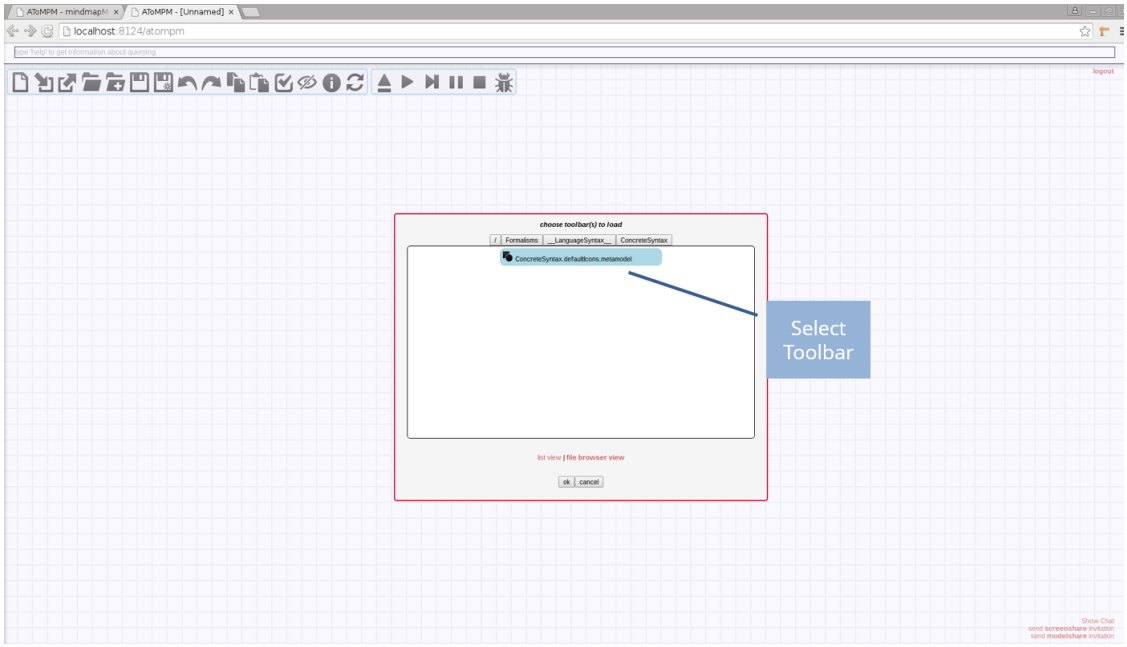


Figure VI.8 – Step 9 to create a DSL in AToMPM

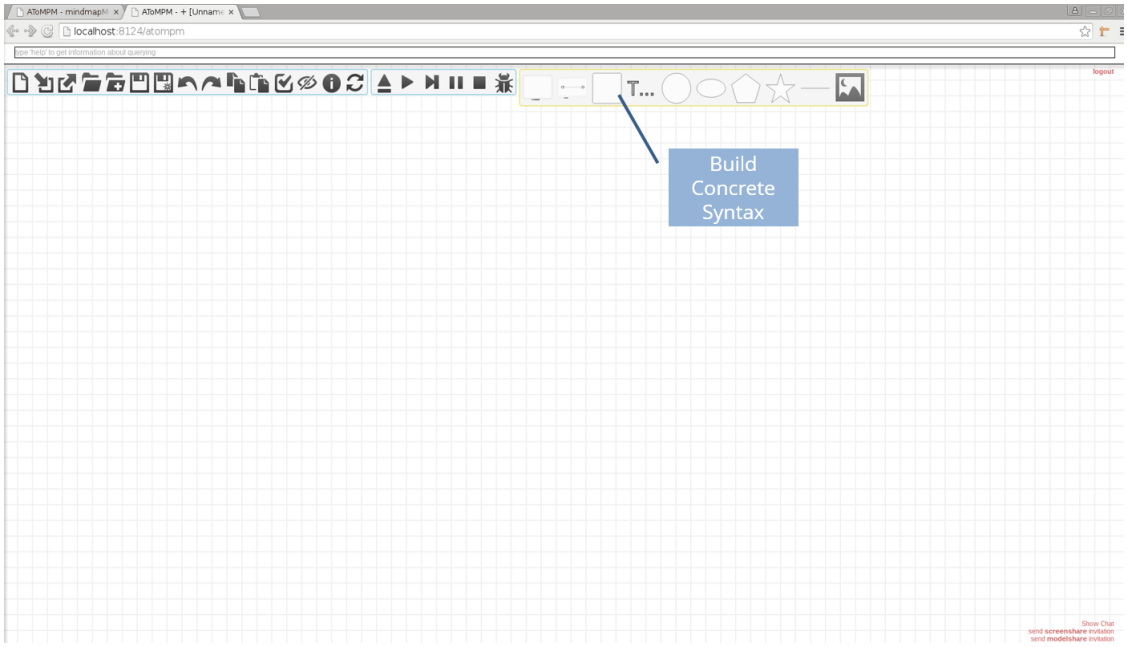


Figure VI.9 – Step 10 to create a DSL in AToMPM

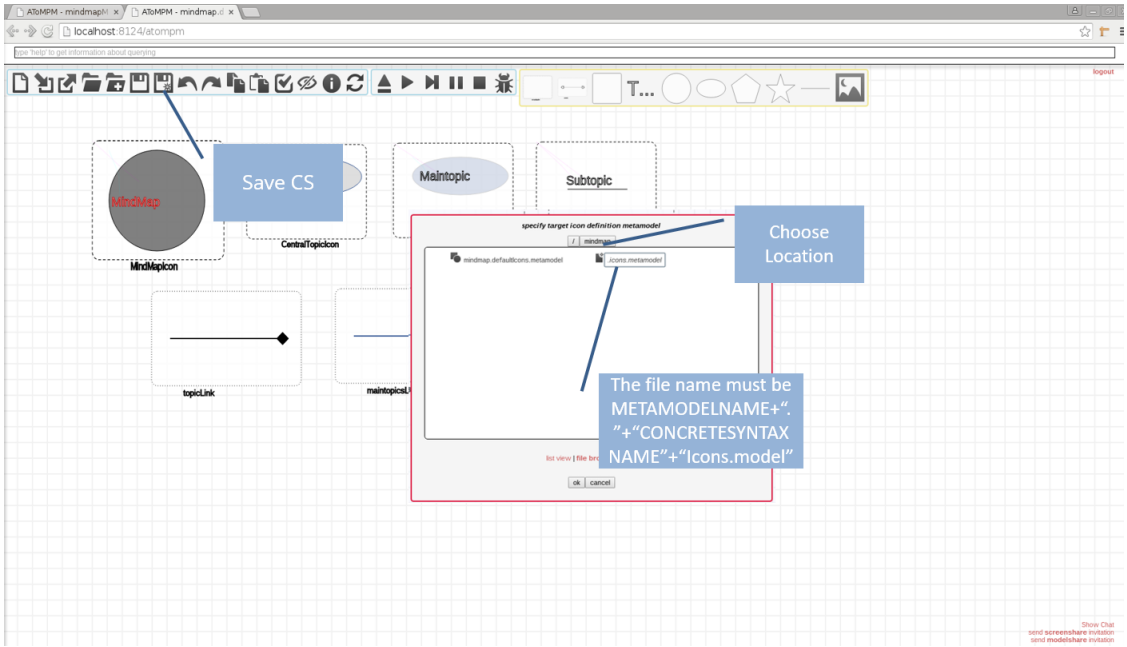


Figure VI.10 – Step 11 to create a DSL in AToMPM

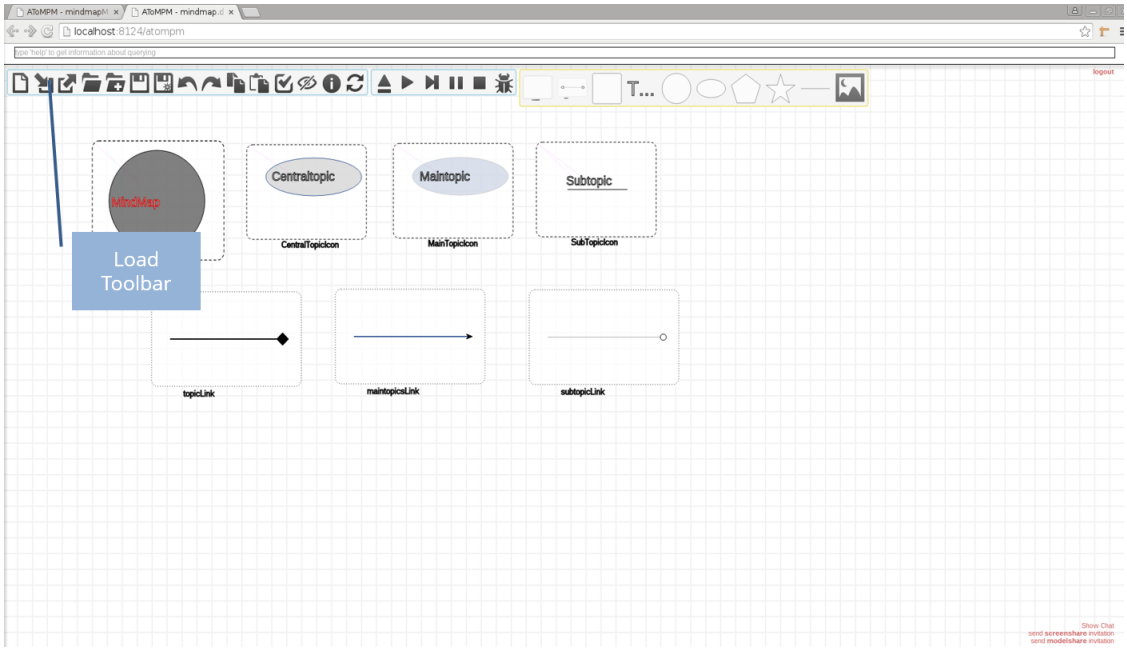


Figure VI.11 – Step 12 to create a DSL in AToMPM

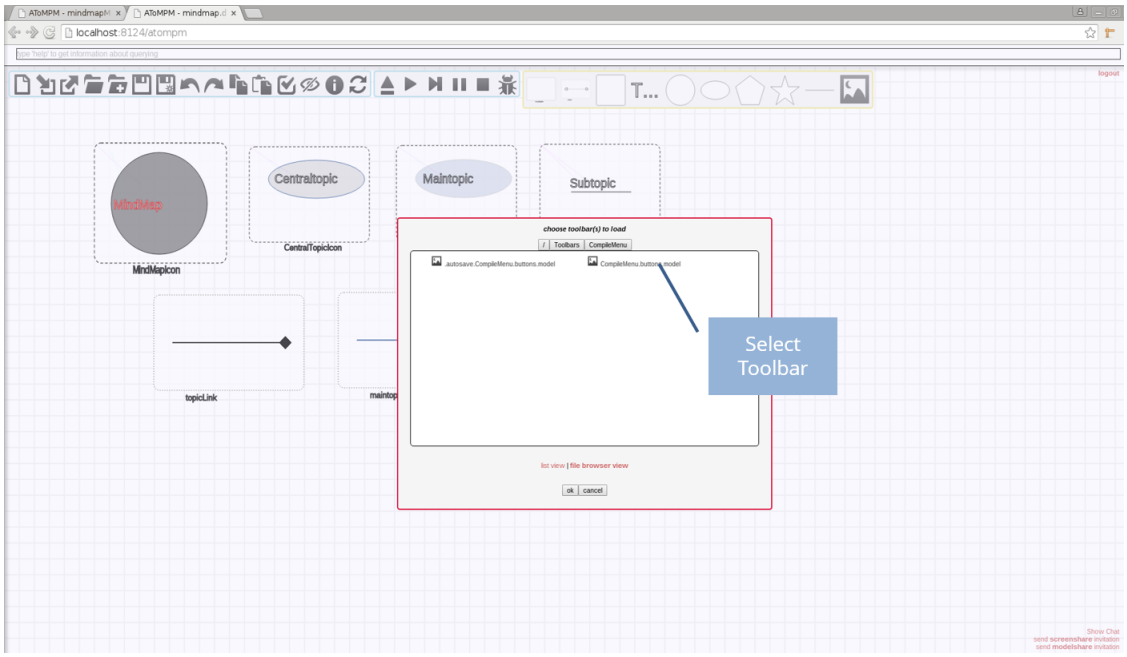


Figure VI.12 – Step 13 to create a DSL in AToMPM

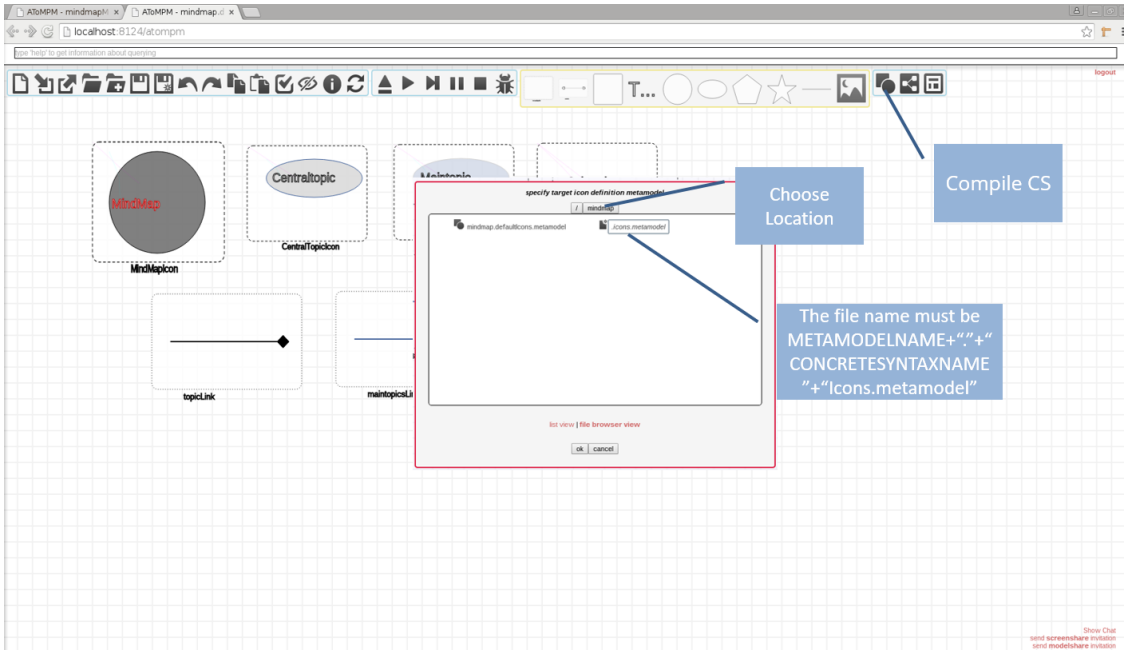


Figure VI.13 – Step 14 to create a DSL in AToMPM

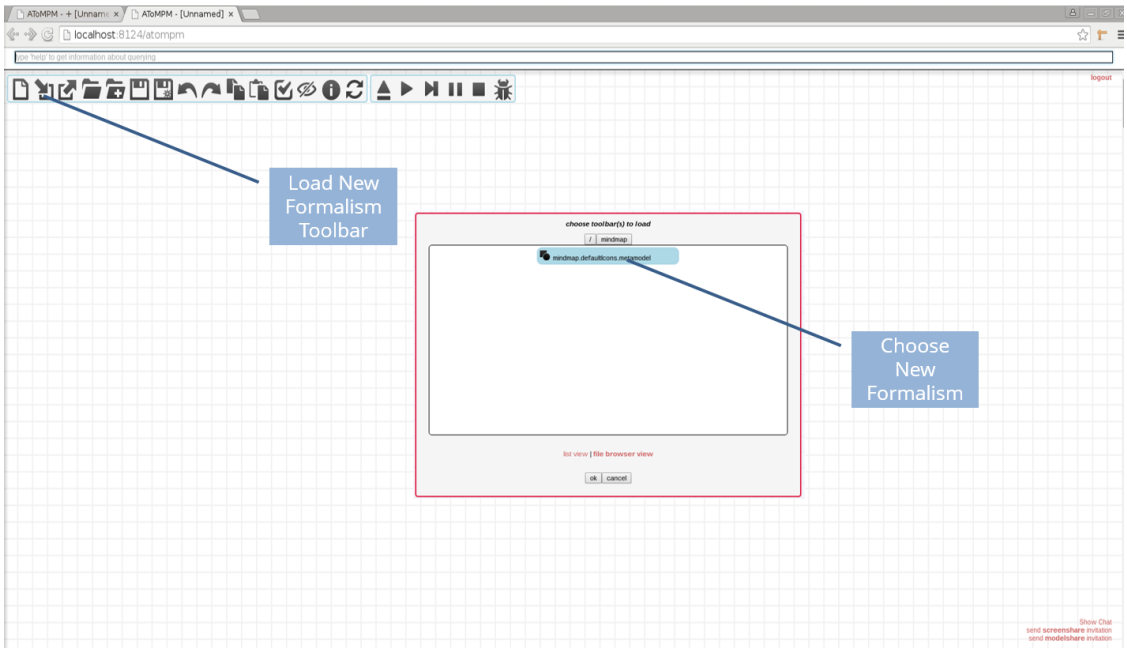


Figure VI.14 – Step 16 to create a DSL in AToMPM

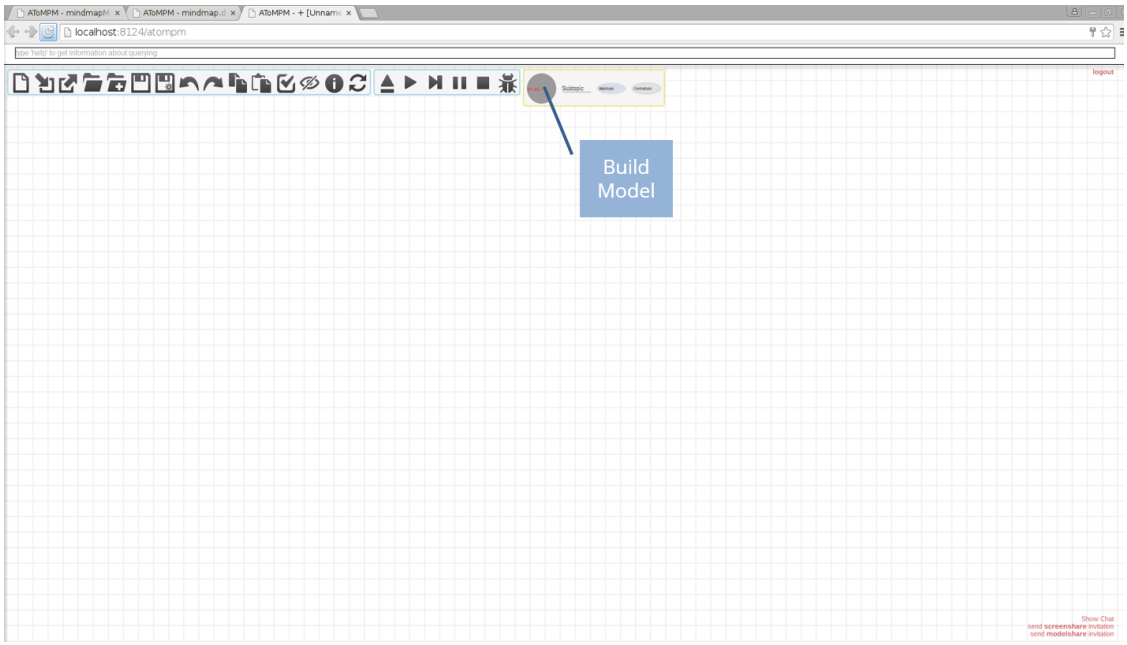


Figure VI.15 – Step 17 to create a DSL in AToMPM

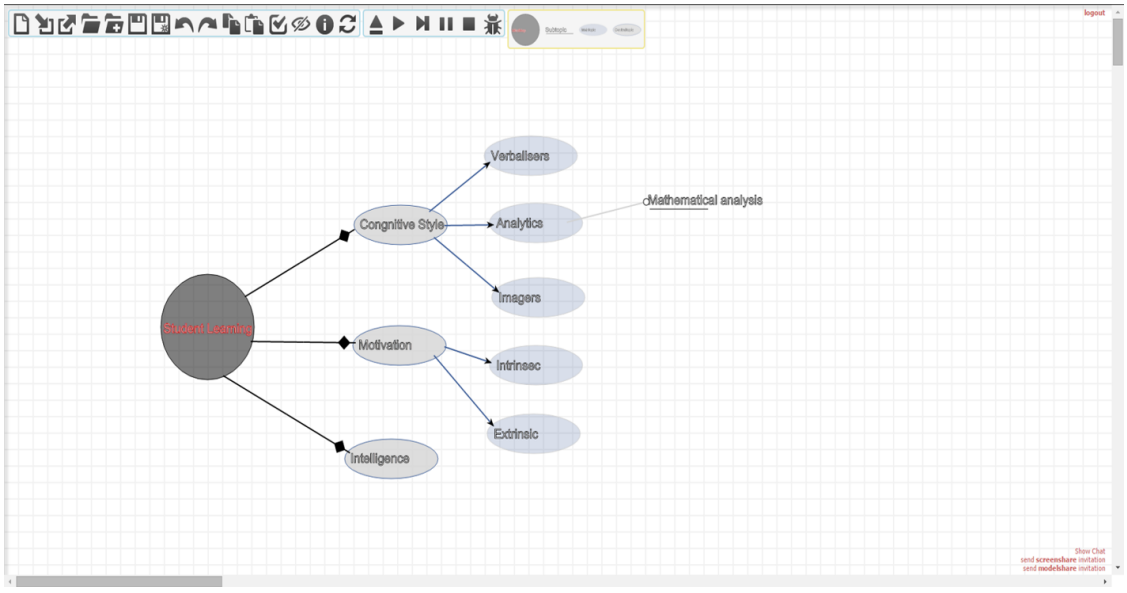


Figure VI.16 – Step 18 to create a DSL in AToMPM

file specifying the textual grammar which indicates how the textual model has to be used. Once completed, he executes the generators to create the domain-specific environment that needs to be launched as a separate tool instance (in this case a new Eclipse instance).

As we can see, 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.

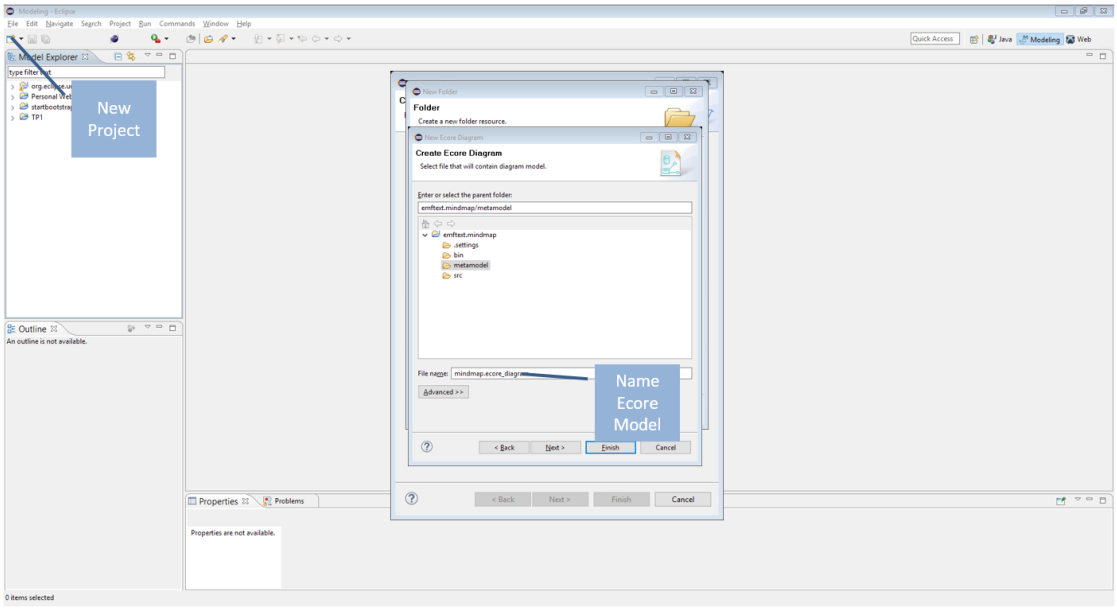


Figure VI.17 – Step 1 to create a DSL in EMFText

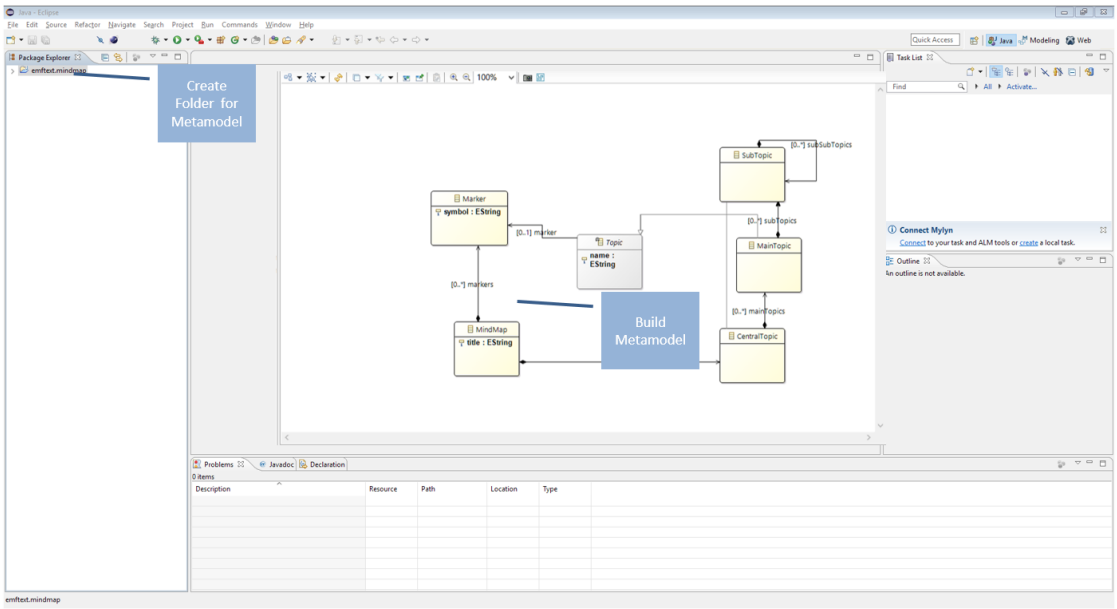


Figure VI.18 – Step 2 to create a DSL in EMFText

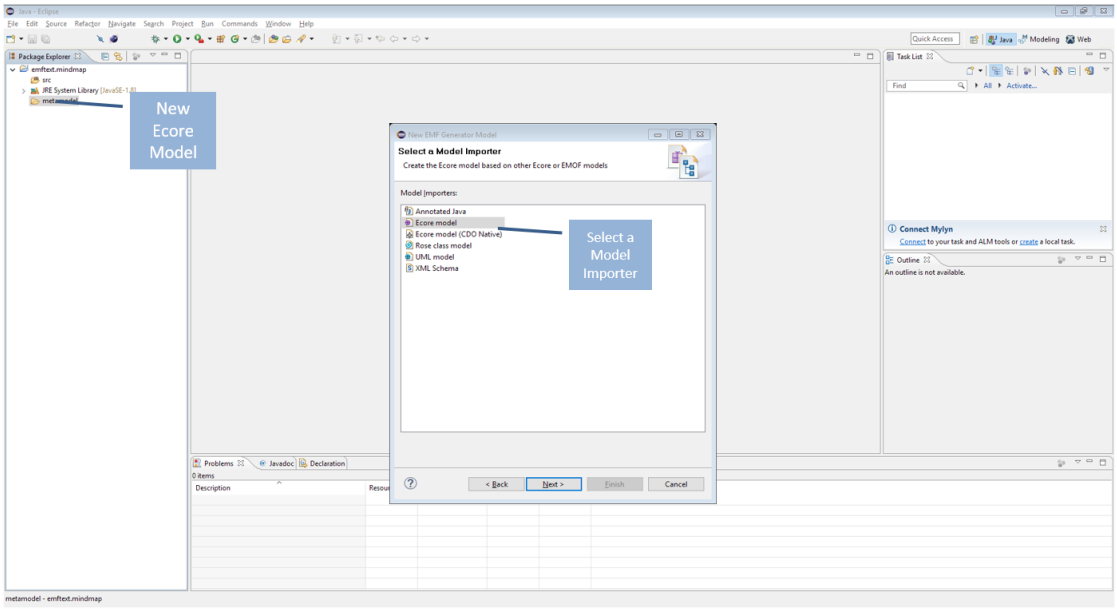


Figure VI.19 – Step 3 to create a DSL in EMFText

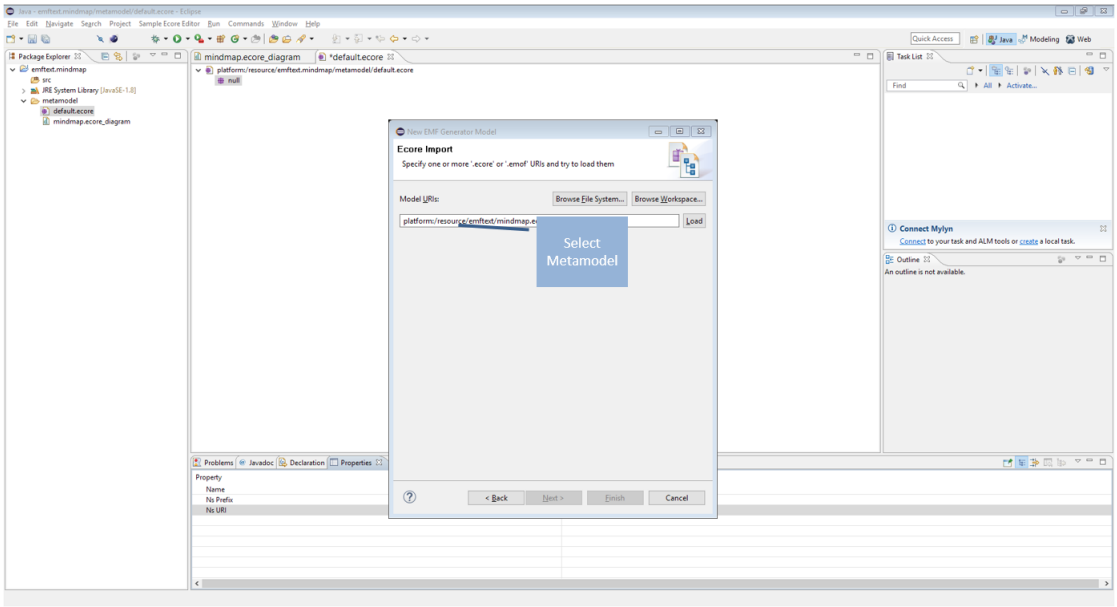


Figure VI.20 – Step 4 to create a DSL in EMFText

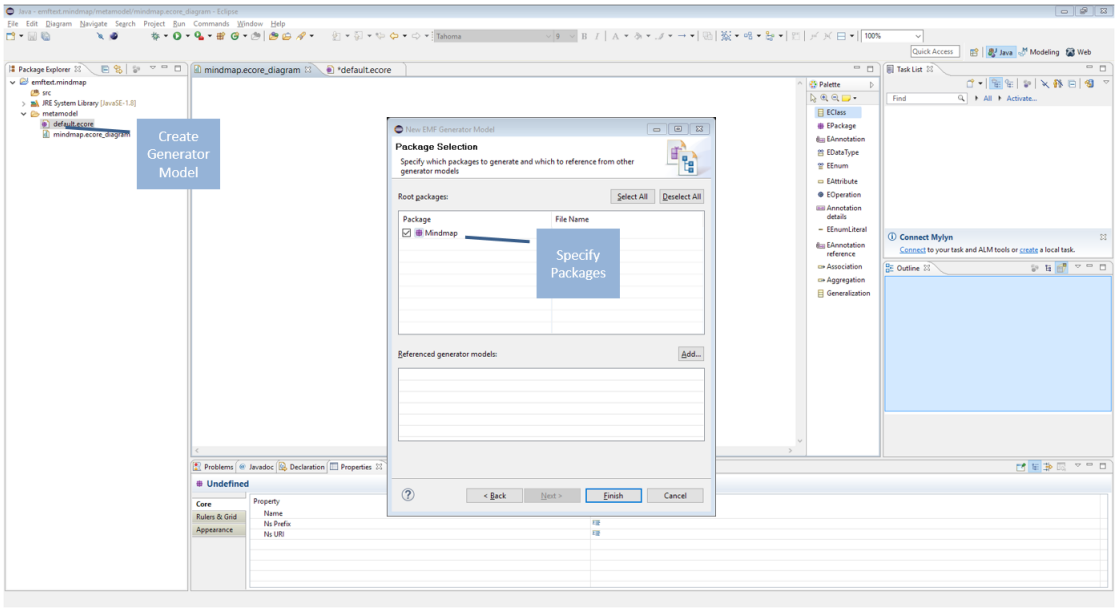


Figure VI.21 – Step 5 to create a DSL in EMFText

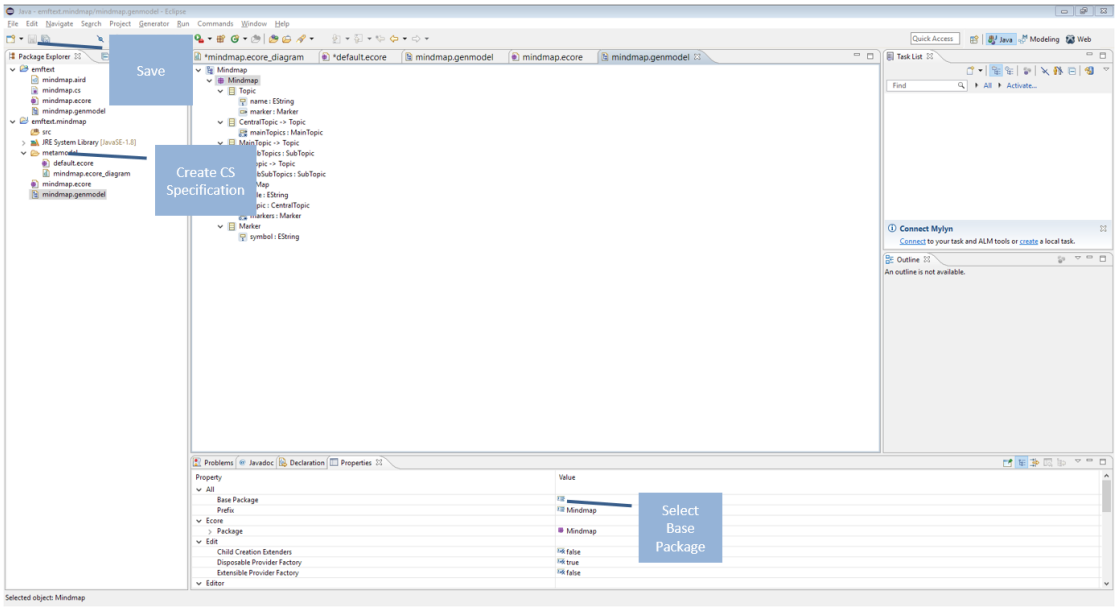


Figure VI.22 – Step 6 to create a DSL in EMFText

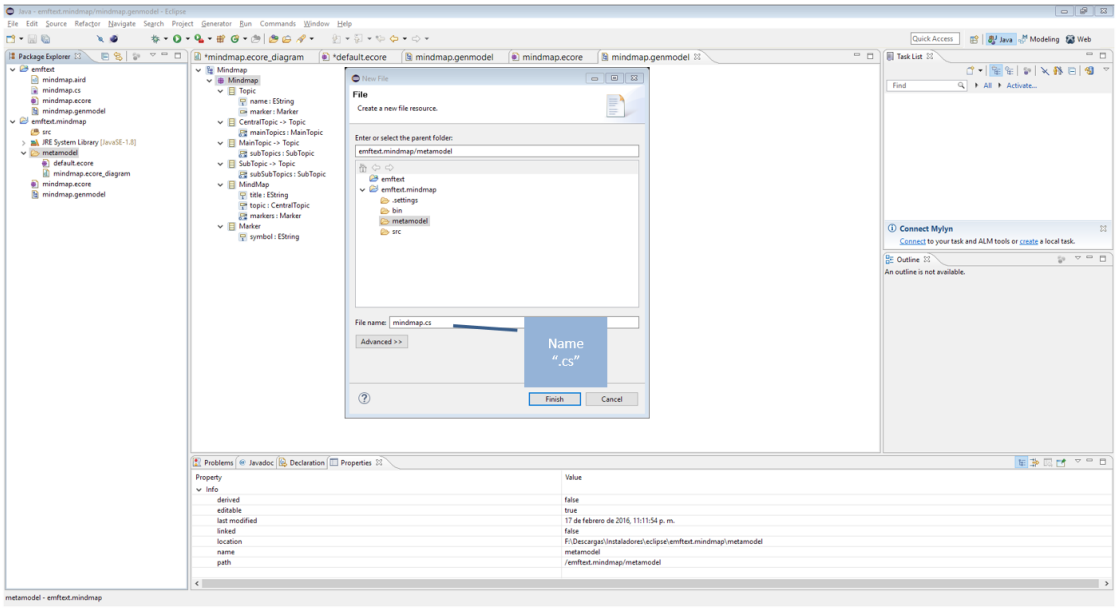


Figure VI.23 – Step 7 to create a DSL in EMFText

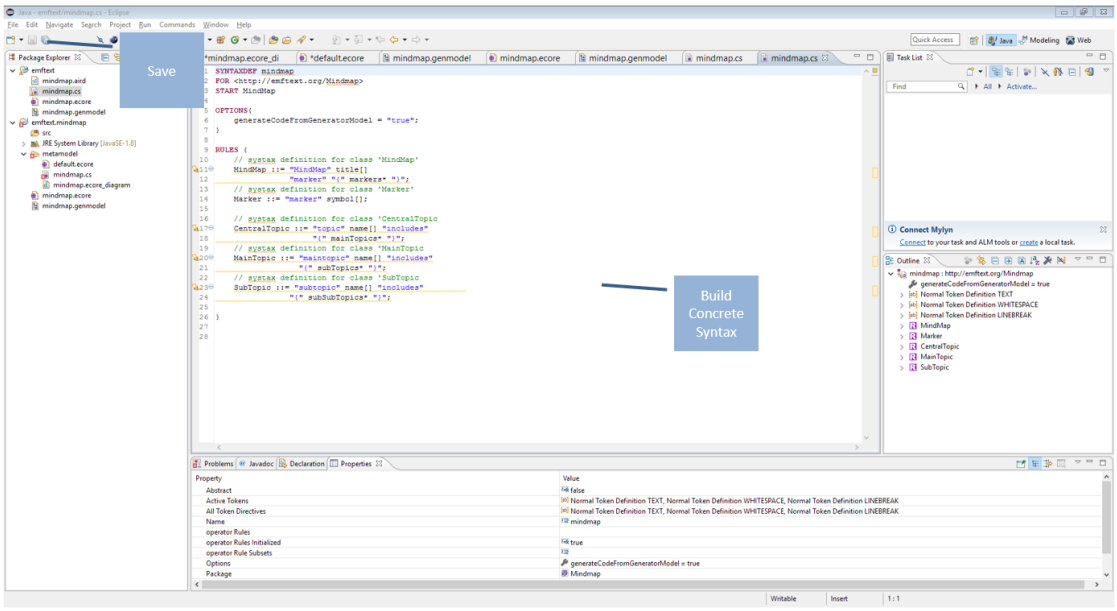


Figure VI.24 – Step 8 to create a DSL in EMFText

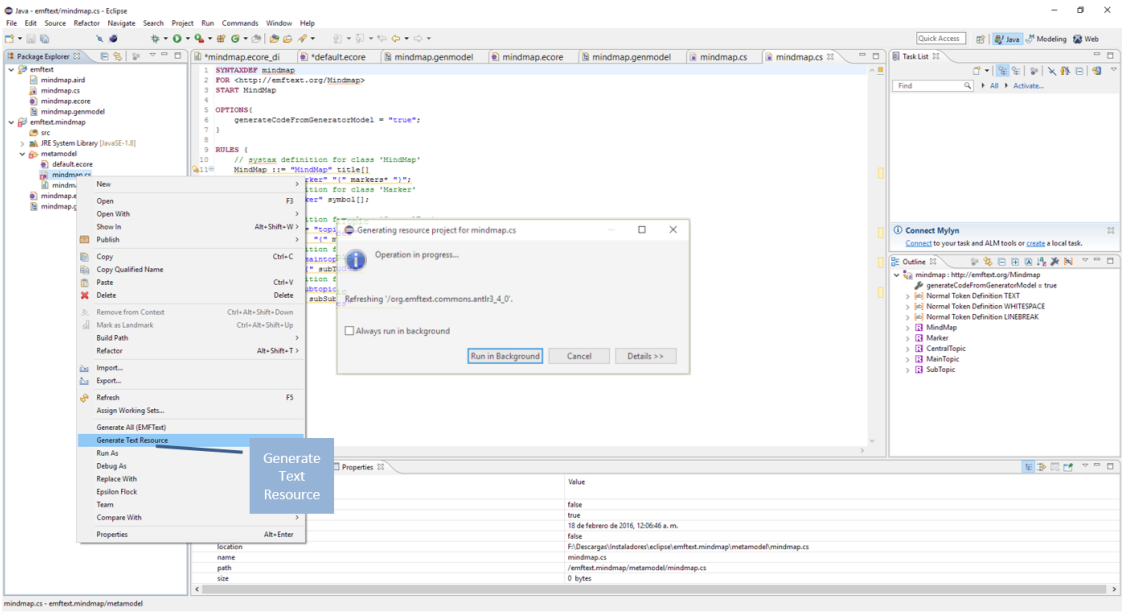


Figure VI.25 – Step 9 to create a DSL in EMFText

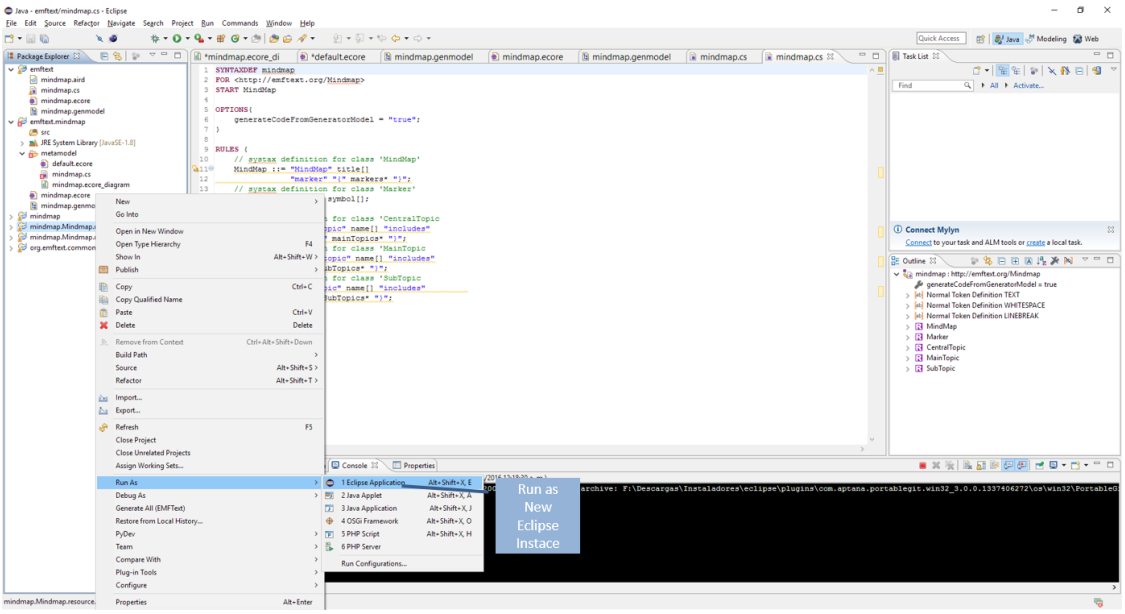


Figure VI.26 – Step 10 to create a DSL in EMFText

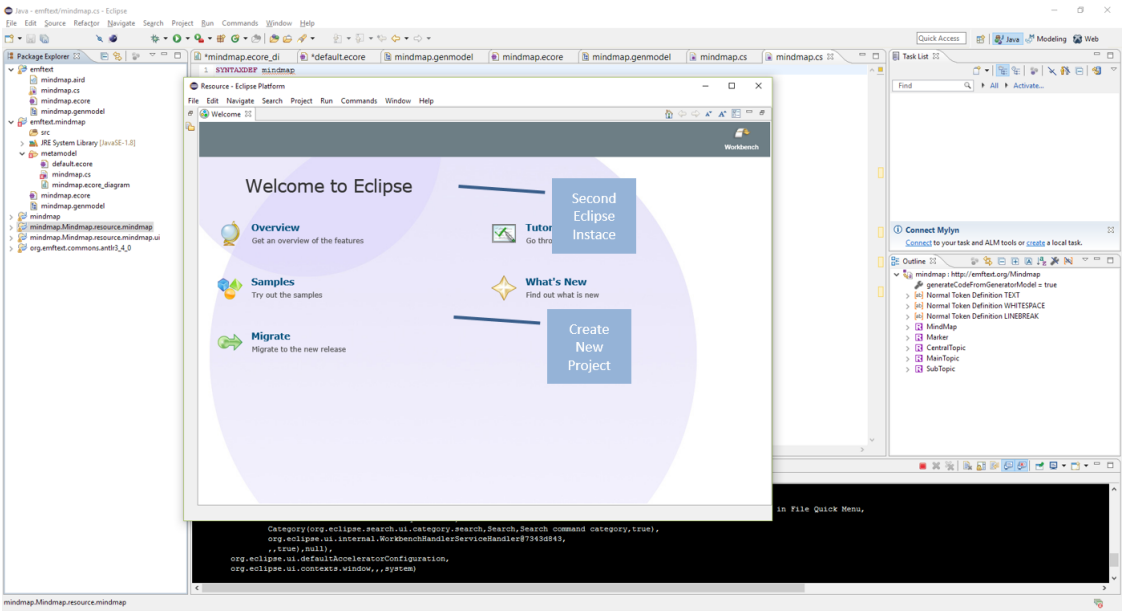


Figure VI.27 – Step 11 to create a DSL in EMFText

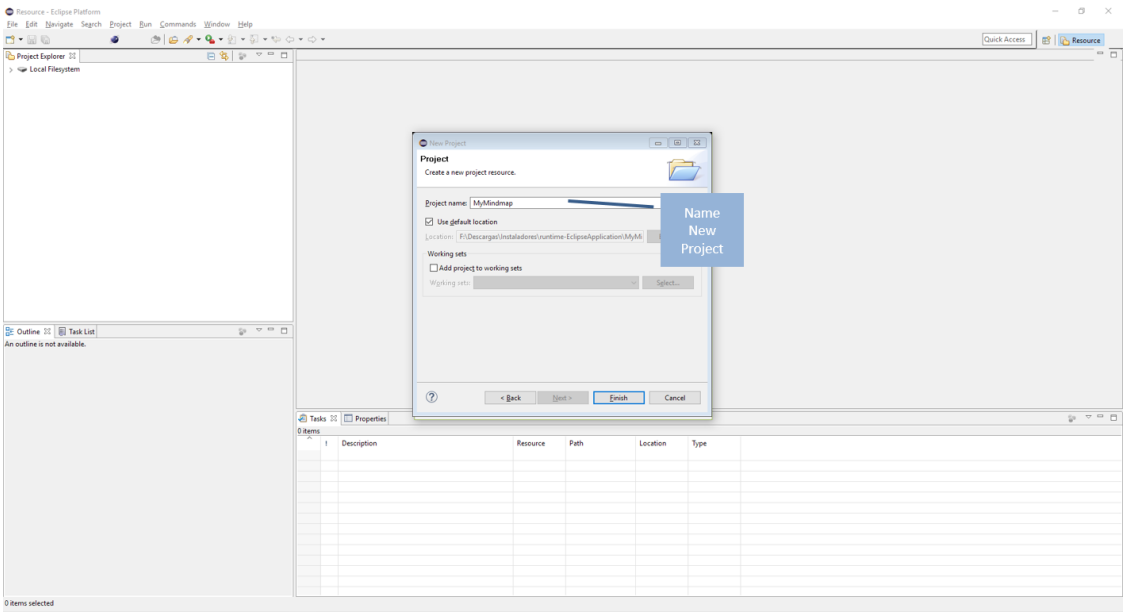


Figure VI.28 – Step 12 to create a DSL in EMFText

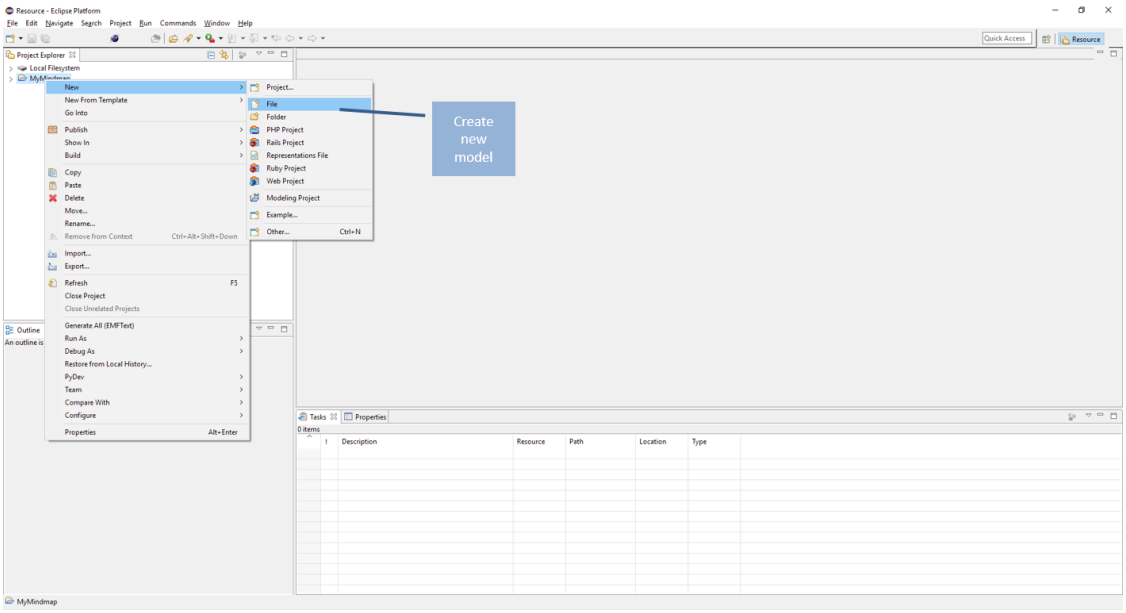


Figure VI.29 – Step 13 to create a DSL in EMFText

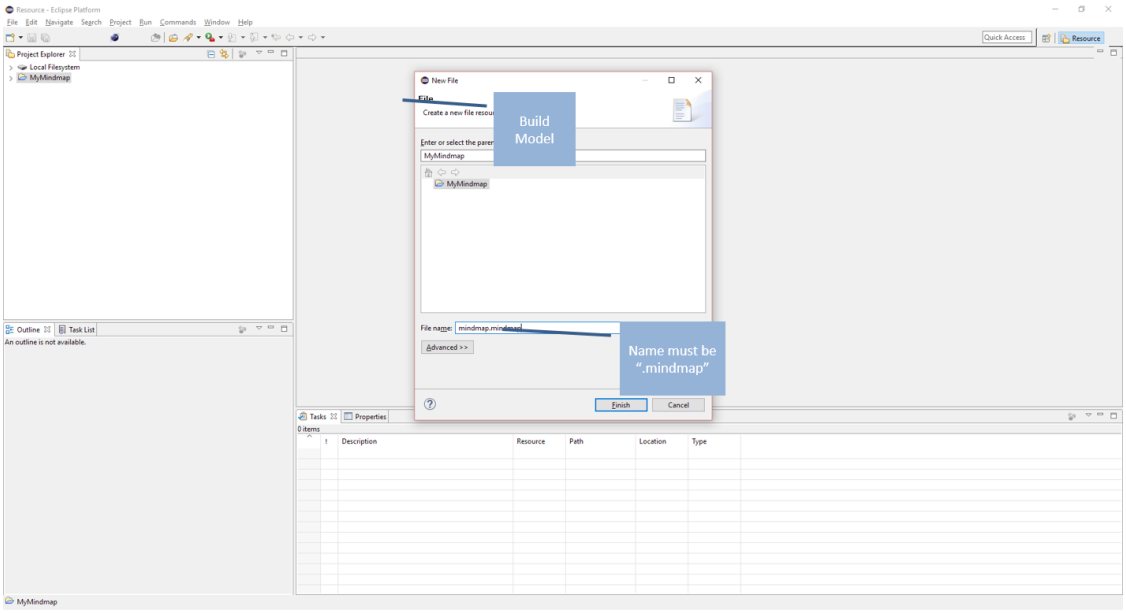


Figure VI.30 – Step 14 to create a DSL in EMFText

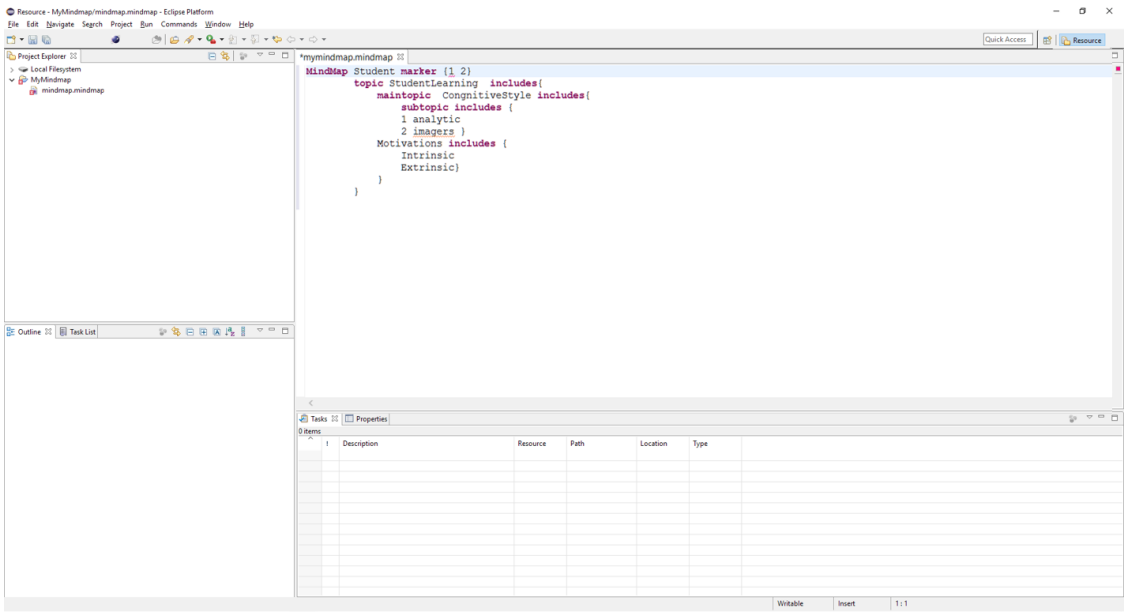


Figure VI.31 – Step 15 to create a DSL in EMFText