Automatic Production of End User Documentation for DSLs

Gwendal Le Moulec, Arnaud Blouin, Valérie Gouranton, Bruno Arnaldi

Automatic Production of End User Documentation for DSLs

Gwendal Le Moulec[a], Arnaud Blouin[a], Valérie Gouranton[a], Bruno Arnaldi[a]

{*firstname.lastname*}*@irisa.fr*

[a]*INSA Rennes, IRISA/Inria, France*

Abstract

Domain-specific languages (DSL) are developed for answering specific problems by leveraging expertise of domain stakeholders. If DSLs are usually small, their development requires a significant software engineering effort: editors, code generators, *etc.*, must be developed to make a DSL usable. Documenting a DSL is also a main and time-consuming task to promote it and address its learning curve. Recent research works in software language engineering focus on easing the development of DSLs. In this work, we focus on easing the production of documentations of DSLs having textual concrete syntaxes. We specifically focus on the automatic production of documentations for DSLs. We adapt from the API documentation domain properties that DSL documentations should follow. Based on these properties we propose a model-driven approach that extracts from DSL artifacts information required to build documentation. Our implementation, called *Docywood*, targets two platforms: *Markdown* documentations for static web sites and Xtext code fragments for live documentation while modeling. We used *Docywood* on two DSLs, namely ThingML and XCore. Feedback from end users and language designers exhibits qualitative benefits of the proposal. End user experiments conducted on *ThingML* and *Target Platform Definition* show benefits on the correctness of the created models when using *Docywood*.

Keywords: software documentation, domain-specific language, model slicing

1. Introduction

A domain-specific language (DSL) is a software language whose expressiveness focuses on a particular domain [1]. DSLs are increasingly being developed to leverage specific domain expertise of various stakeholders involved in the development of software systems [2]. Although DSLs are usually small, their development requires a significant software engineering effort [3, 4]. Concrete syntaxes, editors, compilers are examples of core components of a DSL that must be developed to make it usable. Recent research works focus on easing the development of specific parts of DSLs to reduce their development cost and maintenance [5, 6, 7].

The work proposed in this paper follows this research line with a focus on the user documentation of DSLs. As advocated by Fowler, a "*kind of generator* [that comes with DSLs] *would define human readable documentation - the language workbench equivalent of javadoc.* [...] *There will still be a need to generate web or paper documentation*" [8]. Documenting a DSL is indeed another core and time-consuming development task [1, 3]. This task, however, is required to promote DSLs, address their learning curve [3], and limit the "*language cacophony problem*": because languages are hard to learn the use of many languages will be much more complicated than using a single language [8]. By studying the parallels made between DSLs and APIs (*Application Programming Interface*) [1, 9, 10], we identified four challenges that face end user DSL documentations: 1) Documentation must be complete, *i.e.,* all the DSL concepts must have an up-to-date documentation; 2) Documentation has to be contextualized according to the current need of the DSL users; 3) Maintaining documentation over several platforms is a complex tasks that can lead to documentation obsoleteness; 4) Providing code examples to illustrate each concept of a DSL is a time-consuming task.

This paper describes a model-driven approach that allows the automatic production of end user documentations that explain and illustrate the different concepts of a DSL. The proposed approach focuses on textual and grammar-based DSLs. The approach produces end user documentation from artifacts of the implementation phase of DSLs: the metamodel, the grammar, and models that cover all the concepts of a DSL. For each concept of a DSL, the metamodel, the grammar, and a model are sliced [11, 12] to keep their elements that focus on this concept. A documentation unit dedicated to this concept can be then produced and is composed of: an illustrative example; explanations about the concept and possible parameters. These explanations are not fully synthesized by our approach: metamodel documentation is extracted from the metamodel to be used in the generated documentation. One benefit of the approach is its ability to capitalize on existing DSL artifacts to produce documentation for different platforms. For example, the documentation are currently generated for two platforms: 1) in a *Markdown* format to be easily integrated in wikis; 2) in *Java* code to be seamlessly integrated in the Xtext [5] editor of DSLs and then provide DSL users with live documentation during the auto-completion. Our generative process follows coverage criteria: the produced end user documentation explains all the concepts of the DSL domain model (*e.g.,* all the classes, attributes, and references of the DSL metamodel).

The proposal has been prototyped in *Docywood*[1], built on top of the Eclipse Modeling Framework (EMF) [13] and XText. We validated the proposal through an experiment that involve 11 subjects and two third-part DSLs, namely *ThingML* and *Target Platform Definition*. The experiment exhibits benefits regarding the correctness of the created models when using the generated documentation in complement to the official one. Feedback from the subjects and languages designers assesses the benefits of the proposal and raises some improvements for future works.

The paper is structured as follows. Section 2 introduces an example used throughout the paper to illustrate the approach. Section 3 explains the approach. Section 4 introduces *Docywood*, the implementation of the proposed approach. Section 5 details the evaluation of the approach. Section 6 discusses related work. Section 7 concludes the paper and gives insights for future works.

2. Problem Statement

First, we present an illustrative example for this paper. Then, we formalize the problem to solve.

2.1. Illustrative example: a DSL for moving robots

We define a simple language called *Robot* for moving robots. This language will be used to illustrate the approach throughout the paper.

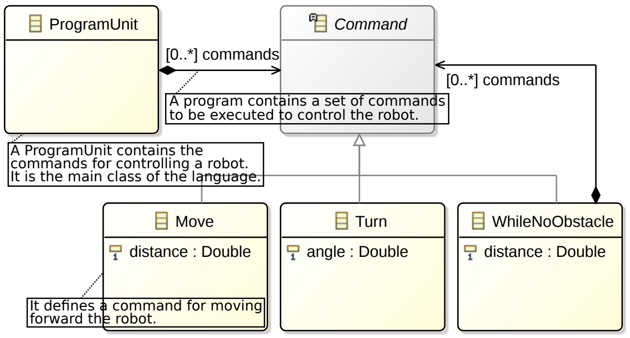

ProgramUnit
Command
[0..*] commands
[0..*] commands
A program contains a set of commands to be executed to control the robot.
A ProgramUnit contains the commands for controlling a robot. It is the main class of the language.
Move
distance : Double
Turn
angle : Double
WhileNoObstacle
distance : Double
It defines a command for moving forward the robot.

Figure 1: The metamodel of the *Robot* language

Figure 1 describes the documented metamodel of the *Robot* DSL. A user can define a program (*ProgramUnit*) to move a robot with commands (*Command*). The commands are: move forward (*Move*); rotate on itself following a given rotation angle (*Turn*); a specific while loop to execute commands while no obstacle are in front of the robot (*WhileNoObstacle*). All the metamodel elements are documented (Figure 1 shows the embedded documentation of three elements, namely *ProgramUnit*, *ProgramUnit.commands*, and *Move*). The documentation, written by language designers, is embedded in the metamodel. For example with EcoreTools, such a metamodel documentation consists of annotations on the metamodel elements.

[1]See: `https://github.com/arnobl/comlanDocywood`

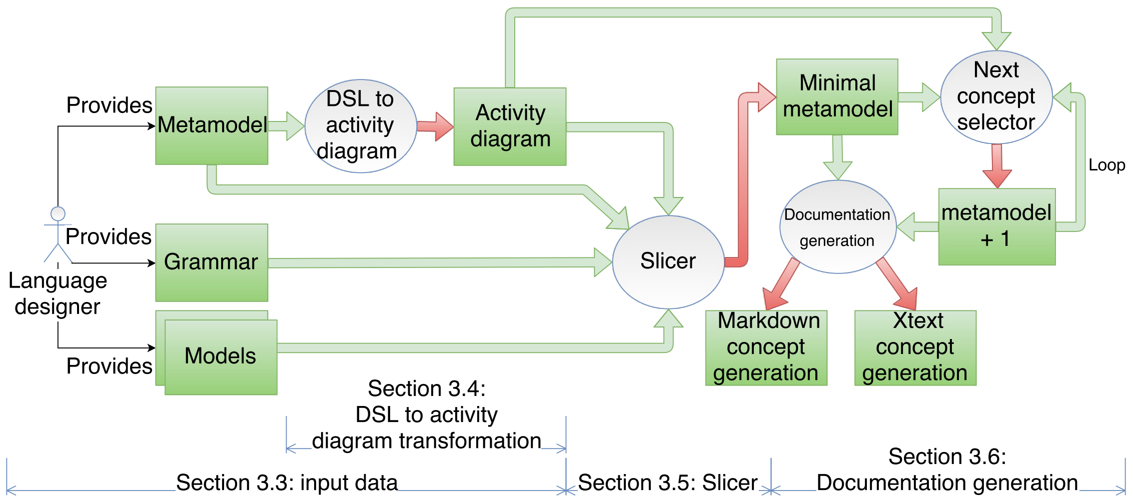


Figure 2: The processing chain of the approach.

Listing 1 shows a *Robot* code snippet that follows the grammar of Listing 2. A *ProgramUnit* surrounds its commands with the *begin* and *end* tokens. The *Move*, *Turn*, and *WhileNoObstacle* commands respectively match the tokens *move*, *turn*, and *whileNoObstacleAt*, followed by their parameter declared between parentheses. A *WhileNoObstacle* command defines sub-commands between brackets.

```
begin
  move(25)
  whileNoObstacleAt(10) {
    move(75)
    turn(90)
    move(50)
  }
  turn(-100)
  move(60)
end
```

Listing 1: A *Robot* model

```
ProgramUnit: 'begin' (commands+=Command)* 'end';
Command: Move | Turn | WhileNoObstacle;
Move: 'move' '(' distance=Double ')';
Turn: 'turn' '(' angle=Double ')';
WhileNoObstacle: 'whileNoObstacleAt' '(' distance=Double
                 ')' '{' (commands+=Command)* '}';
```

Listing 2: The XText grammar of the Robot DSL

2.2. *Problem statement*

Software languages are software too [14] and relations between APIs (*Application Programming Interface*) and DSLs have been studied [1, 9]. So, a parallel can be drawn between DSL and API documentation to precise the problem statement. API documentation pitfalls have been identified from end users feedback [10]. From these pitfalls we define four properties that end users and language designers of DSLs can expect from user end DSL documentation.
Property #1 – Documentation coverage guarantee. Incomplete documentation has been identified as the most important problem that affects API documentation [10]. This property assesses that documentation coverage criteria have been reached. We define the documentation coverage criterion as follows: all the concepts of the DSL are

covered by the documentation. When the domain model of the DSL is a metamodel, this criterion can be decomposed into three coverage criteria: all the attributes, references, and classes of the DSL metamodel are covered by the documentation.

Property #2 – Documentation contextualization. Documentation has to be contextualised according to the current need of the language users. This may limit the *bloat* and *tangled information* issues that consists in providing large and not fully relevant chunks of texts to users [10].

Property #3 – Multi-platform documentation. Manually maintaining documentation over several platforms (several language workbenches, websites, *etc.*) is a complex tasks that can lead to documentation obsoleteness [10].

Property #4 – Example-based documentation. While users generally appreciate examples in documentations, these examples must be supplemented with adequate explanations [10].

In the early stages of the DSL development process, domain analysis can help in identifying the concepts of the DSL and producing some general documentation [15]. Downstream of the development process, language designers can manually write documentation using the current tools such as EcoreTools [13]. These different approaches do not overcome these four properties that characterize the scientific problem of easing the production of DSL documentation: if language designers can document a metamodel created with EcoreTools, this documentation is mainly dedicated to other language designers to build the DSL ecosystem. The four properties concerns the documentation dedicated to *DSL users*, *i.e.,* to users that will use the DSL through its ecosystem (*e.g.,* editors). Moreover, EcoreTools metamodel documentation does usually not come with examples. As highlighted by the property #4, examples must be provided to ease the understanding of DSL concepts. As this task can be time-consuming, an approach must be proposed to ease the production of examples in DSL user documentation.

Based on the four aforementioned properties, this paper proposes a new DSL documentation approach to automate the production of DSL documentation.

3. Approach

This section presents in details the approach. Section 3.1 details the content of the produced documentation. Section 3.2 gives the overview of the approach. The main steps are then detailed in Sections 3.3 to 3.6.

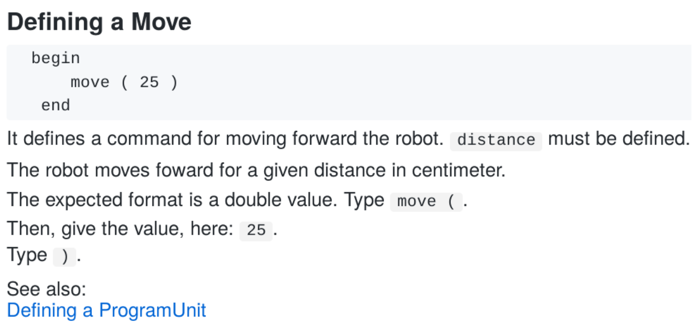
3.1. Documentation Content and Structure

The output documentation produced by the approach is a set of documentation units that document each concept of the DSL under study. We call such documentation units, *concept documentations*.

A concept documentation is composed of: a model in its textual form to illustrate the concerned concept; textual documentation about the current concept; the textual instructions for creating the model using the model as an illustration; references to other related concept documentations.

The model of a concept documentation is extracted from existing models provided as input of the approach. The textual documentation is extracted from the documentation embedded in the metamodel of the DSL. If the DSL under study does not have such metamodel documentation, the approach still works but no explanation about the concepts are provided.

Figure 3 depicts an example of concept documentations for the *Robot* DSL. The concept documentation starts with a model extracted from Listing 1 that covers the targeted concept, *i.e.,* the *Move* command, and its mandatory related elements (here, *ProgramUnit* and the *Move*'s attribute *distance*). Then, the text of the concept documentation explains the different elements not already explained in another concept documentation: We assume that *ProgramUnit* has its own concept documentation so *ProgramUnit* is not explained in the *Move* concept documentation. The text uses: the documentation extracted from the metamodel of Figure 1; the concrete syntax grammar of Listing 2; an illustrative model. Links to the concepts used in the current concept documentation but explained in other ones are provided. For example, the *Move* concept documentation ends with a link to the definition of a *ProgramUnit*.



Defining a Move

```
begin
    move ( 25 )
 end
```

It defines a command for moving forward the robot. `distance` must be defined.

The robot moves foward for a given distance in centimeter.

The expected format is a double value. Type `move (` .

Then, give the value, here: `25` .

Type `)` .

See also:
Defining a ProgramUnit

Figure 3: The concept documentation that explains the *Move* concept of the *Robot* DSL.

3.2. *Overview and scope of the approach*

3.2.1. *Overview*

The proposed approach produces end user documentation for textual DSLs through a model-based approach. Figure 2 depicts the processing chain of the approach. The language designer has to provide as input data to the process the documented metamodel of a DSL, its grammar, and models that will serve as examples in the documentation.

An activity diagram that orders the concept documentations (some concept documentations may depend on other ones, as explained above) can also be provided as input data to the process. This diagram is automatically generated at the first run of the process on a given DSL. Once generated, language designers can edit this diagram to: change the generation order of the concept documentations; merge, remove, or modify concept documentations. Section 3.3 details the characteristics of the input data.

Section 3.4 details the generation of the activity diagram (*DSL to activity diagram* in Figure 2). The process then takes the first activity of the activity diagram to produce the root concept documentation of the DSL. The root concept documentation explains the root concepts of the DSL, *i.e.,* the minimal necessary instructions for a model to be valid.

The production of the root concept documentation follows the steps 3.5 and 3.6 of Figure 2. First, the *Slicer* slices the metamodel to only keep its minimal mandatory elements. Model slicing is an operation that extracts a subset of model elements [11, 12]. Section 3.5 details this module. Second, the *Documentation generation* module uses the root metamodel, the grammar, and the input models to generate the root concept documentation. Then, the root metamodel is augmented with one concept from the original metamodel thanks to the *next concept selector* step of Figure 2. Model slicing is also used during this step to get the mandatory elements related to the concept newly added. The documentation generation and incrementation ("*next concept selector*") steps repeat over the successive sliced metamodels until all the activities of the activity diagram are considered. Section 3.6 details the documentation generation.

3.2.2. *Scope of the approach*

First, the proposed approach detailed in this section focuses on grammar-based and textual DSLs. If graphical DSLs could be supported by the approach, DSLs not based on grammars would require deep changes in the proposed process. Second, the approach relies on the DSL metamodel, grammar, and models to produce end user documentation. The approach does not exploit the DSL behavioral semantics to this end.

3.3. *Input data*

The language designer provides the documented metamodel, grammar, and a set of models of the DSL. A documented metamodel is a metamodel where its concepts (classes, attributes, references, *etc.*) have an associated documentation. For example, elements of an Ecore model (*i.e.,* a metamodel) can be documented. This documentation is embedded in the metamodel as annotations as illustrated in Figure 1. The metamodel has to be documented to provide textual explanations on the concepts explained in the concept documentations. If the metamodel is not documented the process still works but textual explanations will be absent from the generated documentations.

The grammar of a DSL is used to: give instructions in the concept documentations on how to write the current concepts using the concrete syntax; produce code examples from the input models (see Section 3.6).

The input models must cover all the concepts of the DSL. They indeed constitute the examples on which the concept documentations will be based. Their usages are detailed in Sections 3.5 and 3.6.

3.4. DSL to activity diagram transformation

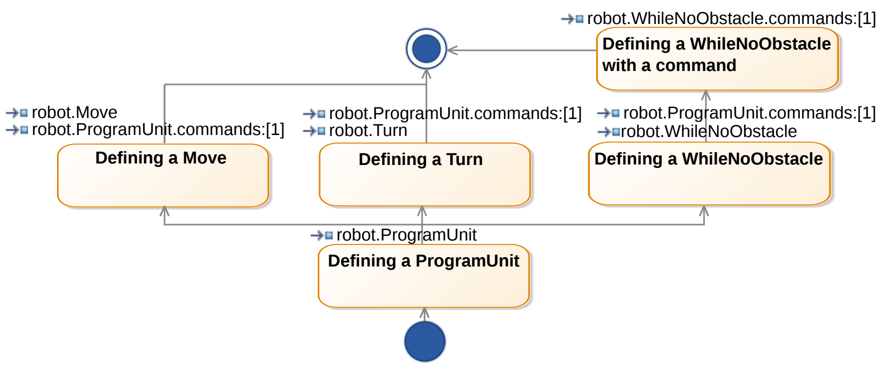

robot.WhileNoObstacle.commands:[1]
Defining a WhileNoObstacle with a command
robot.Move
robot.ProgramUnit.commands:[1]
robot.ProgramUnit.commands:[1]
robot.Turn
robot.ProgramUnit.commands:[1]
robot.WhileNoObstacle
Defining a Move
Defining a Turn
Defining a WhileNoObstacle
robot.ProgramUnit
Defining a ProgramUnit

Figure 4: The generated *Robot* UML activity diagram

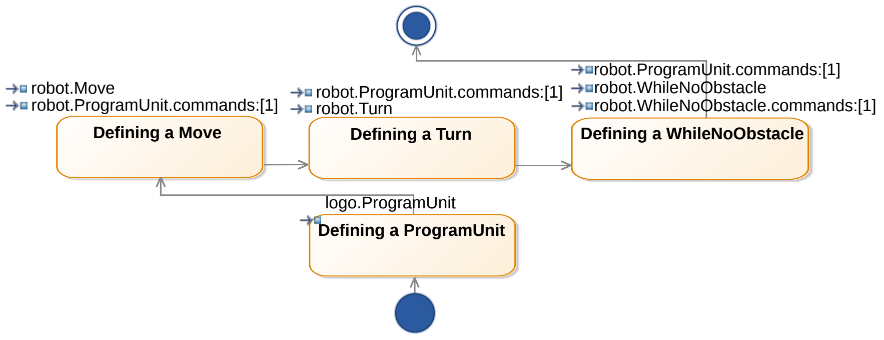

robot.ProgramUnit.commands:[1]
robot.WhileNoObstacle
robot.WhileNoObstacle.commands:[1]
robot.Move
robot.ProgramUnit.commands:[1]
robot.ProgramUnit.commands:[1]
robot.Turn
Defining a Move
Defining a Turn
Defining a WhileNoObstacle
logo.ProgramUnit
Defining a ProgramUnit

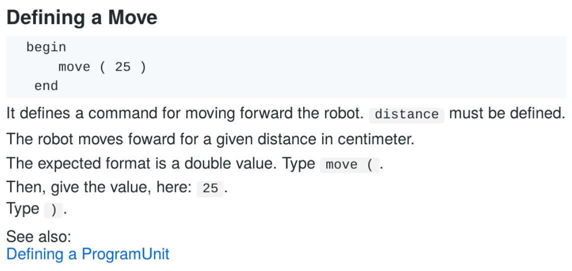
Figure 5: The UML activity diagram of Figure 4 modified by a language designer

The first time a language designer runs the approach for a given DSL a UML activity diagram that describes the sequences of the concept documentations is automatically produced. The goals of this activity diagram are twofold. First, its orders the generation of the documentation. For example, Figure 4 is the UML activity diagram automatically produced for the *Robot* DSL. The *Move* concept is explained after the *ProgramUnit* concept as there is a composition relation between these two classes in the *Robot* metamodel (see Figure 1). This order is mandatory as a generated concept documentation will contain details of the ones previously generated. For example, Figure 6 is the generated documentation for the *Move* concept. The code example contains details about the *ProgramUnit* (the *begin* and *end* tokens) and the text contains references to *ProgramUnit*.

Second, it allows language designer to change this order of generation. For example, Figure 5 is the UML activity diagram of Figure 4 after several changes done by a language designer. The language designer decided to explain *Turn* after *Move*. As a result, the code example of Figure 7 contains the code of the concepts *Move* and *ProgramUnit*.

3.4.1. Activity diagram generation

Algorithm 1 details the generation of the activity diagram. The algorithm follows the structure of the metamodel. The first step consists in detecting the root class of the metamodel (Line 2). The activity diagram is then initialized with an *init node* (Line 4). The activity corresponding to the root class is then produced (Line 5). This activity is always the first activity of the diagram. During the creation of an activity, the current metamodel element to explain is associated to the activity to get it back during the generation of the concept documentation (see *activity.getData()*, Line 6 in Algorithm 2). All the metamodel elements strongly connected to this current element (*e.g.,* an attribute with a lower cardinality of 1) are sliced (Line 10). These elements will be explained in the same concept documentation (if not

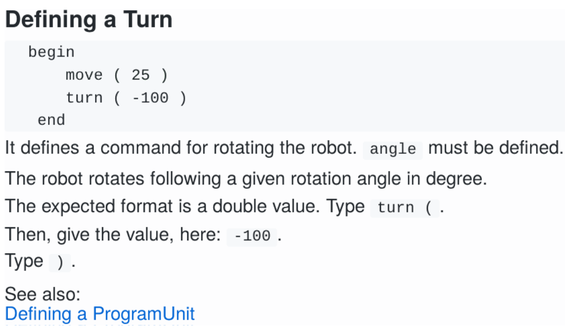


Defining a Move

```
begin
    move ( 25 )
end
```

It defines a command for moving forward the robot. `distance` must be defined.
The robot moves foward for a given distance in centimeter.
The expected format is a double value. Type `move (` .
Then, give the value, here: `25` .
Type `)` .
See also:
Defining a ProgramUnit

Figure 6: A concept documentation that explains the *Turn* concept of the *Robot* DSL



Defining a Turn

```
begin
    move ( 25 )
    turn ( -100 )
end
```

It defines a command for rotating the robot. `angle` must be defined.
The robot rotates following a given rotation angle in degree.
The expected format is a double value. Type `turn (` .
Then, give the value, here: `-100` .
Type `)` .
See also:
Defining a ProgramUnit

Figure 7: A concept documentation that explains the *Turn* concept of the *Robot* DSL

Algorithm 1 Activity diagram generation algorithm

```
1:  procedure GENERATEACTIVITYDIAGRAM($metamodel$)
2:      $rootClass := extractRootClass(metamodel)$
3:      $initNode := new\ InitNode$
4:      $activityDiag := \{initNode\}$
5:      $genActivity(rootClass, metamodel, initNode, \emptyset, activityDiag)$
6:      return $activityDiag$
7:
8:  procedure GENACTIVITY($elt, metamodel, prevActivity, explained, diag$)
9:      $activity := createActivity(elt)$
10:     $explained := explained \cup sliceMetamodel(metamodel, elt)$
11:     $diag := diag \cup \{prevActivity \rightarrow activity\}$
12:     $elements := getElements(elt) \setminus explained$
13:     for each $e \in elements$ do
14:         $genActivity(e, metamodel, activity, clone(explained), diag)$
```

already explained in another concept documentation)[2]. The activity diagram is then completed with the new activity (Line 11). Finally, all the elements referenced by the current element *elt* (*e.g.,* a class has attributes and references, a reference has a target class) and not explained in generated concept documentations are gathered (Line 12). For each of these elements, an activity will be recursively produced following the current activity (Lines 13 and 14). Abstract class are ignored by the algorithm (*e.g.,* the *Command* class). Instead, the children classes are directly loaded.

The algorithm follows coverage criteria to ensure that all the elements of a DSL are explained in one concept documentation: all the attributes, references, and classes of the DSL metamodel are covered by at least one concept documentation. Lines 12 to 14 of Algorithm 1 ensure these coverage criteria by recursively producing activities for elements referenced by the current element.

Circular references are supported: all the classes already encountered are ignored. All the metamodel elements encountered are indeed registered in a set (Line 10) and then discarded from the set of elements to be covered by the activity currently produced. If all elements were encountered, this last set is empty and then the recursive call ends (Lines 13 to 14).

3.4.2. Activity diagram usage

Once the activity diagram has been generated, a language designer can merge, remove, modify activities as well as change the order of the activities. The goal is to let language designers customize the concept documentations to generate. For example with *Robot*, the language designer may want to explain the *Turn* class after and using the *Move* class. Figure 5 is the UML activity diagram of Figure 4 modified by a language designer. The activity dedicated to

[2]The fact that the attributes with a lower cardinality of 0 are not explained in the same documentation is a design choice. We prefer to provide the most simple documentation possible before providing the details.

Turn is now an output activity of the *Move* activity. The *Turn* concept documentation now contains a *Move* command and a reference to the *Move* concept documentation as depicted by Figure 7. This *Move* command is however not explained in this concept documentation but in its own one. The *WhileNoObstacle with a command* activity has also been removed to be explained in the *WhileNoObstacle* activity. So, the reference *WhileNoObstacle.commands* is now explained in this last concept documentation.

By default, an activity diagram that respects our coverage criteria is generated. We think that a language designer can also define its own activity diagrams to design specific documentations, such as tutorials. This point, however, goes beyond the scope of this paper.

3.5. Slicer

A concept documentation focuses on a single metamodel element plus its mandatory related elements. To extract these elements from the original metamodel, a *model slicing technique* [11, 12] is used. Model slicing is a model comprehension technique inspired by program slicing [16]. The process of model slicing involves *extracting* from an input model a subset of model elements that represent a *model slice*. Slicing criteria are model elements from the input model that provide entry points for producing a model slice. The slicing process starts by slicing the input model from model elements given as input (the slicing criteria). Then, each model element linked (*e.g.,* by inheritance or reference) to a slicing criterion is sliced, and so on until no more model elements can be sliced.

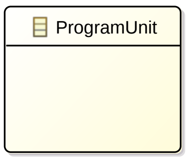


Figure 8: The minimal metamodel of Figure 1 from *ProgramUnit*

Figures 8 and 9 illustrate this slicing process using the *Robot* metamodel (Figure 1). Let us consider that the documentation algorithm currently focuses on the class *ProgramUnit*, the root class of the *Robot* metamodel. The class *ProgramUnit* is considered as the slicing criterion. This class does not have any mandatory element, *i.e.,* no element (reference or attribute) with a minimum cardinality of at least 1. As a result, the slicing of the *Robot* metamodel using the *ProgramUnit* class as the slicing criterion gives the metamodel of Figure 8: the single class *ProgramUnit*. This means that the concept documentation of *ProgramUnit* will detail *ProgramUnit* only.

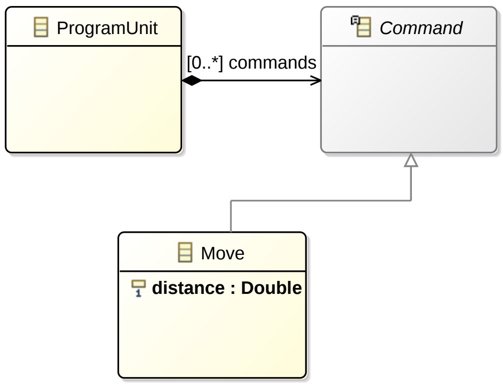


Figure 9: The minimal metamodel of Figure 1 from *ProgramUnit* and *Move*

Now, let us consider that the documentation algorithm focuses on the *Move* class of the *Robot* metamodel. The *Move* class can be explained only after *ProgramUnit* since this last is the root class of the metamodel. So, the model slicer will take as input the slicing criteria composed of the *ProgramUnit* and *Move* classes. Figure 9 is the resulting sliced metamodel. The *Move* class has a mandatory attribute, *distance*, that is sliced. The slicing process recursively slices all the mandatory elements. This means that in a same concept documentation, several concepts of the DSL can be explained if they are interdependent.

For each concept documentation to produce, a slicing process is performed to obtain the corresponding sliced metamodel. From this sliced metamodel, a model that has the required elements is then sliced to illustrate the current elements only. Only the elements of the sliced metamodel are explained in the text that supplements the illustrative model.

3.6. Documentation generation

This section details the documentation generation process.

3.6.1. General process

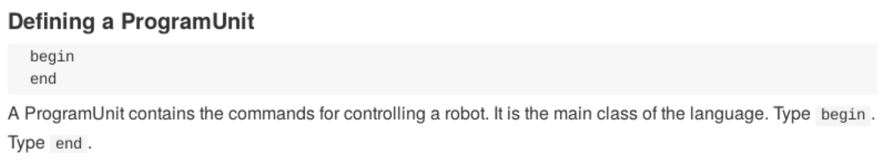
The documentation generation step takes as input the metamodel, the grammar, the set of models, and the UML activity diagram. Algorithm 2 details the generation of the concept documentations. The documentation generation is driven by the UML activity diagram (*cf.* Section 3.4 for the automatic production of the activity diagram at the first run of the approach): one activity implies one generated concept documentation, where the activity is tagged with the concepts (attributes, references, or classes) to explain and their origin (*i.e.,* the source reference). As detailed in Lines 12 and 13, once an activity has been treated, its output activities are recursively treated to produce the next concept documentations.

Algorithm 2 Documentation Generation

```
Require: metamodel, grammar, models, activityDiagram
 1: prevElts := ∅
 2: rootActivity := getRootActivity(activityDiagram)
 3: produceDoc(rootActivity, prevElts, metamodel, models, grammar)
 4:
 5: procedure PRODUCEDOC(activity, prevElts, mm, models, grammar)
 6:     elements := findElements(activity.getData(), mm)
 7:     inputSlicer := prevElts ∪ {elements}
 8:     slicedMM := sliceMetamodel(mm, inputSlicer)
 9:     slicedModel := findAndSliceModel(slicedMM, models)
10:     produceMarkdownText(slicedMM, slicedModel, prevElts, grammar)
11:     prevElts := prevElts ∪ {elements}
12:     for each nextActivity in activity.getOutputs() do
13:         produceDoc(nextActivity, clone(prevElts), mm, models, grammar)
14:
15: procedure PRODUCEMARKDOWNTEXT(slicedMM, model, prevElts, grammar)
16:     codeExample := fromXMItoConcreteSyntax(model)
17:     actions := extractActions(slicedMM, grammar)
18:     text := toText(actions, codeExample)
```

For example, Figure 4 represents the *Robot* UML activity diagram produced at the first run of the approach. The generation of the documentation starts with the root activity of the activity diagram (Lines 2 and 3), *e.g.,* the activity “*Defining a ProgramUnit*” for the *Robot* activity diagram. The metamodel elements associated to the activity (*i.e.,* the class *ProgramUnit*) are obtained (Line 6). Then, model slicing is used to get the sliced metamodel to explain. All the previous metamodel elements already explained are collected in *prevElts*. The set *prevElts* and the elements of the current activity are gathered, in *inputSlicer*, to be provided as input of a model slicer (Lines 7 and 8). This model slicer then slices the original metamodel to produce as output the minimal metamodel than contains at least all the elements of *inputSlicer*. For example with the root class *ProgramUnit*, Figure 8 depicts the sliced metamodel.

The text of the concept documentation is then produced using this sliced metamodel (Line 10). Finally, all the output activities of the current one are recursively treated (Lines 12 and 13). For example, according to the UML activity diagram of Figure 5, the next activity is the “*Defining a Move*” activity, which goal is to explain the *Move* class. The input of the model slicer is now the previous element *ProgramUnit* plus the *Move* class. Figure 9 depicts the output sliced metamodel. The concept documentation is then produced from this metamodel. The elements explained in previous concept documentations are not explained again. For example, the class *ProgramUnit* is not explained in the concept documentation that focuses on *Move*. The process goes on with the two last activities, namely "*Defining a Turn*" and "*Defining a WhileNoObstacle*".


Defining a ProgramUnit

```
begin
end
```

A ProgramUnit contains the commands for controlling a robot. It is the main class of the language. Type `begin`. Type `end`.

Figure 10: A concept documentation that explains the root element, *i.e., ProgramUnit*, of the *Robot* DSL

3.6.2. *Markdown and Xtext generation*

The Markdown generation of a given concept documentation is summarized Lines 15 to 18 of Algorithm 2. The model that will illustrate the concept documentation is produced by: 1) finding a model, among those provided by the language designer, that has all the elements of the sliced metamodel to explain; 2) slicing the found model to match the current sliced metamodel (Line 9); 3) converting the sliced model into its concrete syntax to be integrated in the text (Line 16). Then, the grammar is analysed to extract the sequence of typing actions to perform to write the model (Line 17). For example, given the *Robot* grammar (Listing 2) and the root activity "*Defining a ProgramUnit*", Figure 11 depicts the corresponding sequence of actions. The grammar rule corresponding to the *ProgramUnit* class is the first rule (Line 1 in Listing 2). It corresponds to three actions: typing the keyword "*begin*", then a list of commands, and then the keyword "*end*". In the sliced metamodel (see Figure 8) the *Command* class corresponding to the second action does not exists. This action is thus ignored. So, the resulting concept documentation is only composed of the actions "*begin*" and "*end*". The resulting *Markdown* concept documentation is shown in Figure 10. The two first sentences of the text come from the metamodel documentation.

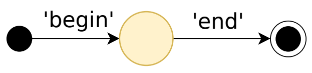


Figure 11: The sequence of generated actions for the activity "*Defining a ProgramUnit*" (see Figure 4)

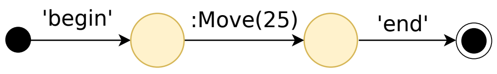
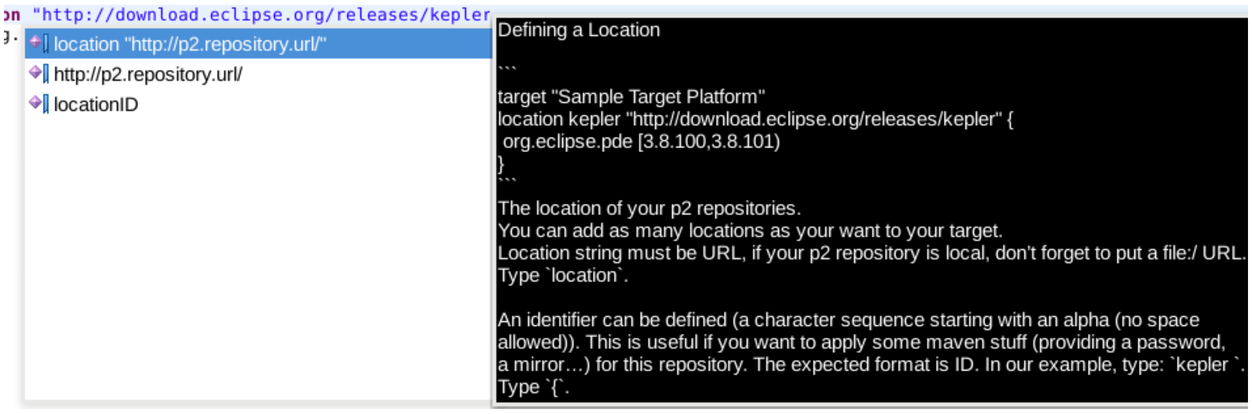


Figure 12: The sequence of generated actions for the activity "*Defining a Move*" (see Figure 4) and the example model of Listing 1

Given the second activity "*Defining a Move*" and the model of Listing 1, Figure 12 depicts the sequence of actions. The same reasoning applies. Contrary to the previous example, the *Command* class exists in the sliced metamodel (see Figure 9). So, the corresponding action is explained. The only available alternative of the second rule of the grammar (Listing 2) is *Move*. So, only the first *Move* command of the example is taken into account in the concept documentation: *move(25)*. Figure 3 shows the output *Markdown* concept documentation for this activity. The documentation embedded by the metamodel elements is used to provide the textual explanations. Given the *Robot* example, the process goes on with the activities "*Defining a Turn*", "*Defining a WhileNoObstacle*", and "*Defining a WhileNoObstacle with a command*" that follow the same process that for "*Defining a Move*".

By default, the process uses a metamodel element one time in a concept documentation, *e.g.,* the concept documentation of Figure 3 contains a single *move* command. Language designers can change the cardinality in the activity diagram (*e.g.,* by changing a "[1]" to a "[2]" in Figure 4). This implies that the input models have the requested cardinalities.

The approach also generates an Xtext fragment that can be integrated into the Xtext editor of the DSL. The goal of this fragment is to contextualize the documentation while an end user is modeling, as depicted by Figure 13. The Xtext fragment generation integrates the generated Markdown documentation into the API Xtext.



~~~
on "http://download.eclipse.org/releases/kepler
}.
location "http://p2.repository.url/"
http://p2.repository.url/
locationID
~~~

~~~
Defining a Location

```
target "Sample Target Platform"
location kepler "http://download.eclipse.org/releases/kepler" {
 org.eclipse.pde [3.8.100,3.8.101)
}
```
The location of your p2 repositories.
You can add as many locations as your want to your target.
Location string must be URL, if your p2 repository is local, don't forget to put a file:/ URL.
Type `location`.

An identifier can be defined (a character sequence starting with an alpha (no space
allowed)). This is useful if you want to apply some maven stuff (providing a password,
a mirror…) for this repository. The expected format is ID. In our example, type: `kepler`.
Type `{`.
~~~

Figure 13: Example of contextualized documentation in an Xtext editor

4. Tooling

We implemented the approach detailed in Section 3 in a tool named *Docywood*. This tool is built on top of the Eclipse Modeling Framework (EMF) [13]. *Docywood* also relies on XText [5] to manage the description of the textual syntaxes of DSLs. The model slicing technique used in the approach is performed by Kompren, a DSL for defining model slicers [11, 12]. *Docywood* takes as input an Ecore model (the metamodel of the DSL), instances of this metamodel (models of the DSL), and the XText model that defines the textual syntax of the DSL. *Docywood* can also takes as input the UML activity diagram that describes the documentation to generate. If not provided as input, this diagram is automatically generated during the execution of *Docywood* to let language designers be able to edit it for next documentation generations. To illustrate the ability of the proposal to target different platforms, *Docywood* can generated documentation in: the Markdown format to be integrated on web sites as static documentation; Xtext code fragment to be seamlessly integrated in the completion of the Xtext editor of the DSL and provide live documentation. The coverage criteria currently supported by *Docywood* are all the classes and all the attributes. This means that *Docywood* may not produce documentation for all the references. For example with *Robot*, the current version of *Docywood* does not explain the reference *WhileNoObstacle.commands* since *Commands* already have dedicated concept documentations.

5. Evaluation

We evaluate qualitatively and quantitatively the benefits of the proposal by conducting an empirical experiment on real third-part DSLs. We first describe the experimental protocol to then detail the results of the experiment. Discussions conclude the evaluation. The materials of this experiment are available on the companion web page of this paper[3].

5.1. Experimental design

The goal of the experiment is to empirically observe whether the generated documentations have, quantitatively and qualitatively, a beneficial impact on the use of DSLs. We decompose this global research question in three specific *research questions*:

RQ1 Does the use of the generated documentations improve the correctness of those tasks of DSLs?

RQ2 Does the use of the generated documentations reduce the time needed to complete typical tasks of DSLs?

RQ3 Do end users consider the generated documentations as beneficial for the understanding of DSLs?

[3]See: `https://github.com/arnobl/comlanDocywood`

RQ4 Do language designers consider the approach valuable for documenting DSLs?

The *objects* of the experiment are two DSLs. The first DSL is ThingML, a DSL for modeling embedded and distributed systems [17]. The second DSL is TPD (*Target Platform Definition*) [4] dedicated to the definition of Eclipse target platforms (*i.e.,* Eclipse plug-in repositories). These two DSLs have been selected using the following criteria: they are developed by third parties (ThingML is developed at SINTEF, Norway and TPD by Obeo/Eclipse, France); they are developed on top of EMF and Xtext; they both have online documentation[5]; models of these DSLs can be used; they have different characteristics (the ThingML metamodel has 96 meta-classes while TPD has eight meta-classes).

We used *Docywood* on two DSLs before the experiment: the *Robot* DSL we designed to illustrate this work; The *Kompren* DSL, a language for defining model slicers (Kompren is also used in the approach itself to slice models) [11, 12]. We do not use Kompren in the experiment since one of the authors of this paper is a maintainer of this DSL. The generated documentations for these DSLs are available on the Kompren's web page[6] and on the companion web page of this paper.

Tasks. We designed two exercises, one for ThingML and one for TPD. The ThingML exercise consists of modeling a component that receives random values from a port and that sends each obtained value to a screen display from another port with alternatively the color black or red. This exercise has been validated by a ThingML language designer.

The TPD exercise consists of modeling a target platform that imports from different other sources specific plug-ins with specific versions for a given platform.

Dependent variables. The quantitative research questions RQ1 and RQ2 require the following dependent variables:

CORRECT To answer RQ1 we manually evaluate the models created by the subjects while performing the tasks. This measure is done in percentage of correctness. We manually corrected the exercises.

TIME To answer RQ2 we measure the time spent by the subjects to perform the modeling tasks using ThingML and Xcore.

Subjects. 11 subjects did the experiments: one master student, six PhD students, one engineer, and three researchers. They all have a background in software engineering and are recurrent users of different DSLs. Their skills in ThingML and TPD were asked just before the experiment to organize the experimental procedure.

Procedure. Each subject did alone the ThingML and TPD exercises in a random order. Half of the subjects did the ThingML exercise (resp. TPD exercise) using the official documentation of the DSL only. These subjects did the TPD exercise (resp. ThingML exercise) using the official documentation plus the documentation generated by *Docywood*. The goal is not to state whether the generated documentation is better than the official one, but whether it improves the modeling tasks of DSL users. Before the experiment of each subject, we explained them how to use the documentations. The subjects had no way, but the type checker of the DSL editor, to test or check their models.

The documentation generated by *Docywood* and provided to the subjects is composed of the Markdown and Xtext documentations. the Markdown documentations were put on the companion page of the paper. the Xtext code fragments were integrated into the editor of each DSL. Two versions of each DSL editor were used during the experiments: one with our Xtext code fragment integrated and another one without it.

To answer RQ3, we asked for anonymous feedback from the subjects (as end users). For RQ4, we sent an email to the language designers of the two DSLs.

[4] `https://github.com/mbarbero/fr.obeo.releng.targetplatform`

[5] See `https://github.com/TelluIoT/ThingML` for ThingML and the official web page of TPD.

[6] `https://github.com/arnobl/kompren`

5.2. Analysis and results

The results of the experiments are analyzed and discussed in this section. Regarding the quantitative results, we apply the independent samples Mann-Whitney tests [18] (data do not follow a normal distribution) to compare the performance, in terms of time and correctness, that may bring the documentations generated by *Docywood*, using a 95% confidence level (*i.e., p*-value < 0.05).

Dependent variables	**Mean Official + *Docywood***	**Mean Official**	**Mean Diff**	**Significance *p*-value**
$CORRECT_{THINGML}$	96%	70%	**+26%**	**0.043**
$CORRECT_{TPD}$	92.5%	95%	- 2.5%	0.504
$TIME_{THINGML}$	31.44m	33.47m	-2.03m	0.841
$TIME_{TPD}$	9.54m	10.03m	-0.09m	0.930

Table 1: Results of *CORRECT* and *TIME* on ThingML and TPD with their official documentation compared to their official documentation plus the one generated by *Docywood*.

RQ1. Regarding the correctness of the models designed by the subjects (Table 1), the average correctness measured shows a benefits when using the documentations generated by *Docywood* on ThingML (+26%). This results is significant (*p*-value < 0.05). The average correctness measured on TPD shows no significant results. To conclude on RQ1, the generated documentation improves the correctness of the models designed by end users of large DSLs (like ThingML).

RQ2. Regarding the time spent by the subjects to perform the exercises (Table 1), the average time measured shows no significant benefits when using the generated documentation. We think that for large DSLs (such as ThingML), users need to understand the goal of the language before doing the exercise by browsing the official documentation. This is not the goal of our generated documentation that rather focuses on helping users while modeling with a DSL. To conclude on RQ2, the generated documentation does not help newcomers of DSLs to do their first models faster. The time spent by the subjects to do the basic ThingML exercise (around 30 minutes) shows that the initial learning curve is a challenge to tackle in future works.

RQ3. Several points were anonymously evaluated (between 1 and 5, 1 is useless, 5 is mandatory / very good) by the subjects at the end of their experiment: the global usefulness of the generated documentation; the usefulness of the code examples; the readability of the documentation; the completeness of the documentation; the usefulness of the coding instructions in the documentation. The results are detailed in Figure 14. Subjects were then invited to give free and anonymous comments.

Several lessons can be drawn from these data. We link them to the end user DSL documentation properties described in Section 2.2. First, the subjects globally appreciated the documentation and found it useful and complete to perform the exercises (property #1: documentation coverage guarantee). Regarding the pros of the generated documentation, one subject appreciated that the documentation integrated in Xtext is "*very direct*" (property #2: documentation contextualization). The documentation is also "good to remind me or to describe me how to do something". "*I am quite surprised by the readability* [of the documentation]."

Second, several cons have be detailed: "*the generated documentation is not self-contained*"; "*I am not sure that is could be a good 'getting start' documentation*". The goal of the generated documentation is not to replace the official one that usually contains how-to-start documents, tutorials, or global explanations about the goal of the language.

Third, the code examples provided in each generated documentation unit were strongly appreciated (property #4: example-based documentation). We think that providing users with example for each concept of a DSL is a crucial point. One user suggested that "*I would also like to have different examples for the same concept*". This is possible using our model slicing approach and may be an input parameter of *Docywood*. One user explained that the "*provided names lack semantics which do not make clearer the code snippets*". This is one point that language designers have to consider while using our approach: the code example are not introduced and language designers must carefully select them to improve the understanding of the documented concepts.

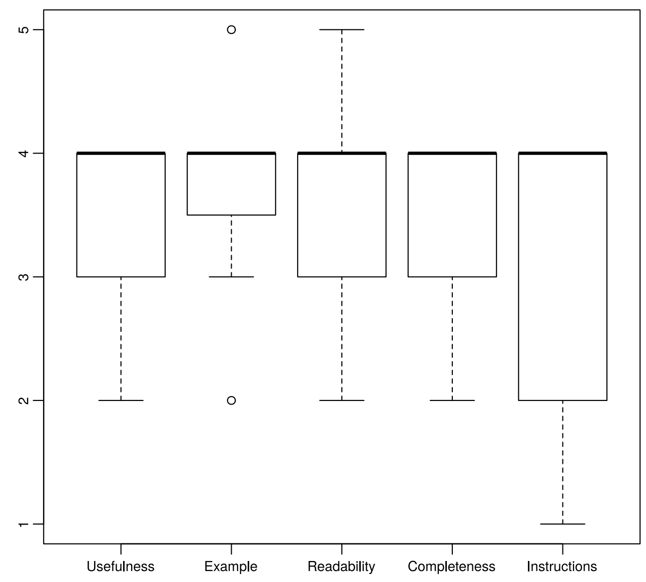


Figure 14: Anonymous evaluations of the generated documentation by the subjects

Fourth, the coding instructions provided in the concept documentations may not be useful. One subject explained that the "*typing instructions may be difficult to follow*". We think that adding such instructions in the documentation should be optional.

Finally, "*TPD is probably too simple to enjoy the generated documentation*". "*The auto-completion is sufficient for this language*". We think that small DSLs, as TPD, may not benefit of our documentation as the keyword of the textual syntax may be sufficient to understand it. This may explain the non-significant results we got on TPD.

To conclude on RQ3, the subjects globally consider the generated documentations as beneficial for the understanding of large DSLs. They identified different improvements to make in future works.

RQ4. ThingML language designers give us feedback regarding the generated documentation (we had no answer from the TPD language designers). They appreciated the generated documentation and consider it as a useful complement of the official documentation that is hard to maintain on different supports (property #3: multi-platform documentation). They noticed that each classes' parameter has, by default, its dedicated documentation. They consider this choice as not always relevant preferring to have a single concept documentation that includes the optional attributes. We explain them that this is possible by editing the UML activity diagram they consider interesting to bring flexibility to the approach. In a next version of *Docywood*, we may add an option to document a class and all its attributes in a single concept documentation.

To conclude on RQ4, and similarly to RQ3, the interviewed language designers consider the approach valuable for documenting DSLs. They identified several current limits, discussed in the next sub-sections, and improvements to make in future works.

5.3. Threats to validity

Internal validity. The obtained results may depend on the quality of the metamodel documentation provided by the language designers. To limit this threat, we selected two DSLs with existing metamodel documentations. The TPD metamodel documentation, however, is more detailed than the ThingML one.

External validity. This threat concerns the possibility to generalize our findings. We designed the experiments using two DSLs with different characteristics and developed by different language designers. Our approach, however, focuses on grammar-based textual DSLs.

Regarding the population validity, we asked the subjects to evaluate their skills for each DSL between 1 and 5 (5 is expert) before the experiment. We use these data to balance the group of users that use a DSL with our generated documentation and the group of users that use the official documentation only. We do not consider *TIME* and *CORRECT* for the subjects that have strong skills in ThingML and TPD as they already know the syntax of the DSL. We, however, got their feedback to discuss RQ3. This concerns one subject with ThingML.

Construct validity. This threat relates to the perceived overall validity of the experiments. We designed representative yet simple exercises to limit the time of the experiment for each subject to limit their tiredness. The feedback from the subjects were gathered anonymously.

5.4. Limit of the approach

The proposal has the following limitations. First, as explained in Section 3.2.2 metamodels do not capture the whole semantics of DSL. Metamodels may be supplemented with constraints declared using the OCL language [7] or within the type checker of an editor. The proposed approach does not support such constraints yet. For example with ThingML, as noticed by its language designers the initial documentation for the *CompositeState* class contained the following code excerpt:

```
composite state c init s1 {
}
```

This code conforms the ThingML metamodel but is not correct because of constraints declared in the Xtext type checker of ThingML: the initial state *s1* must be part of the *CompositeState*:

```
composite state c init s1 {
   state s1 {}
}
```

To overcome this limitation language designers can modify the UML activity diagram produced by the approach. This is what we did with ThingML to have a state *s1* defined in the composite state *c*.

Second, on metamodel changes the documentation and the activity diagram produced by *Docywood* can be re-generated. However, if the activity diagram has been previously modified its changes would be lost. Approaches to the co-evolution of models and metamodels exist and could be used [19, 20].

Third, the ThingML and TPD DSLs did not have a metamodel with embedded documentation. Instead, the classes and attributes of their metamodel are explained on their web site. The reason is that these DSLs use Xtext to generate the Ecore metamodel from the grammar, which does not support Ecore documentation. We had to manually put these documentations into the metamodels. In future work, *Docywood* will consider other sources of metamodel documentation to overcome this drawback.

Fourth, UML activity diagrams may be hard to explore and edit on large DSLs because of the high number of activities created by the approach. This is a usability issue of the UML tools and techniques have been proposed to ease the understanding of large UML models [21].

6. Related work

Software Documentation. Because software languages are software too [14], we detail the major techniques used for documenting software systems. Various approaches have been proposed to automatically infer API documentations or recommendations by analyzing their usages [22]. These approaches require a large data set to analyze. DSLs are dedicated to a specific and thus narrow audience. The quantity of artifacts created from these DSLs may not be large enough to be mined to infer documentations. Manually-authored API documentation may be incomplete or wrong and should be complemented with the output of automated techniques [22]. Our approach follows this claim by proposing an automated documentation process based on coverage criteria.

Software languages, such as UML, can be used as documentation tools of software systems. Arisholm *et al.* shown that the time spent to update UML models, used as documentation, during code changes limits the benefits of this documentation [23]. They argue that UML tools should be improved to support the co-evolution of the code and

[7] `http://www.omg.org/spec/OCL/`

UML models. Our approach goes in that way by allowing developers to re-generate the concept documentations with no maintenance cost on grammar or documentation changes.

Automated exercises production. Complementary to our proposal, the automated production of exercises aims at helping the learning of a language or concepts. The application domains are for example the production of exercises for online courses [24] or to illustrate mathematical problems [25]. In MDE, Gómez *et al.* propose an approach that generates exercises for training users in recognizing well-formed and badly-formed models [26]. To this end, their approach generates models that focus on a targeted concept (*e.g.,* state machines) and mutates them to obtain badly-formed models. Their method, as ours, has the advantage of being metamodel independent. Our method could improve exercise generation by introducing the notion of exercises sequences. In the method of Gómez *et al.*, the exercises are generated in a random order and do not necessarily focus on one concept of the metamodel. Exercise generation could benefit of such concepts, for example to address the metamodel concepts in an order that makes sense in terms of level of difficulty. It would be interesting for language designers to provide some information to guide the exercise generation order, like the UML activity diagram of our approach.

Automated models production. In MDE, model generating and mutating techniques are mainly used to test model transformations. Model transformation testing and documentation production share similarities. In particular, considering coverage criteria is mandatory to measure the elements of a model transformation (or a metamodel) covered by tests (or documentation). Metamodel coverage has been widely studied in the literature [27, 28]. Metamodel coverage can be done at two levels [28]: class coverage and cardinality coverage (*i.e.,* attributes and associations cardinalities). Regarding cardinality coverage, proposed techniques consist in defining partitions over cardinalities (*e.g.,* $[0..*]$ is partitioned to $\{\{0\}, \{1\}, [2..*]\}$) [29, 30]. Then, at least one value of each partitioning set must be tested. The main challenge in class coverage is to limit the number of models (*i.e.,* it is not necessary for the same class to be covered by too many tests). To this end, several methods exist: classifying models in equivalence class and select a representative one [31]; translate this objective to a multi-function optimization problem [32]. As explained in Section 3.4.1, our approach follows the following coverage criterion: each class, attribute, and reference is covered by one concept documentation. Several metamodel elements can be covered in the same concept documentation.

In their approach, Batot *et al.* proposed a framework for generating a minimum number of models that cover a metamodel and that focus one a minimum number of metamodel elements [32]. Their approach is based on a genetic algorithm. This approach can hardly be reused in our context since the models are randomly generated: the input models used to illustrate the concept documentations would not be as understandable as models provided by language designers. The method of Batot *et al.* could be helpful if language designers lack of models for their DSLs and want to quickly produce concept documentations.

Decision Making. The automatic production of documentations is related to the design of auto-completion and related techniques that help users while defining models. Proactive modeling aims at assisting users while defining models [33]. This technique complements ours since it proposes to users actions to do while defining graphical models. Similarly to our proposal, this approach analyses the domain model to extract information about the next actions to propose to the user.

Use Case Modeling and domain analysis. Use case modeling is done to capture functional requirements [34]. Domain analysis can be used to obtain, in the context of software language engineering, domain models (*e.g.,* metamodels) [15]. Approaches have been proposed to transform use cases into domain models [35]. These approaches come in the early stages of the DSL development process to identify the DSL concepts. Our approach comes downstream of the development process to document DSLs based on their developed concrete artifacts.

7. Conclusion and Future Work

7.1. Conclusion

In this paper we proposed an automated approach for producing DSL documentations. The approach has been implemented in *Docywood* and has been evaluated with the ThingML and TPD DSLs. The evaluation exhibits that the proposed approach improves the correctness of models of large DSLs created by novice end users. The evaluation does not exhibit benefits for small and easy to understand DSLs. End users and language designers we interviewed for the evaluation globally agree that the approach is useful in complement of official documentations. They identify several drawbacks such as: coding instructions that may not be useful; the readability of the generated documentations strongly depends on the example selected by the language designers and the readability of the metamodel documentation. Globally, the generated documentation does not aim at replacing the official one. It rather completes the existing tutorials, how-to-start documentation written by language designers.

7.2. Future Work

The current proposal has several limits that we will study in future works. First, we will investigate how to encompass DSLs with concrete graphical syntaxes. We will also focus on the auto-completion to provide users with model skeletons produced from the grammar by using a slicing process. Easing the initial learning effort of new users is also a challenge to overcome: End users need explanations about a DSL and its goals are mandatory to start using this DSL and our generated documentation. Finally, we will investigate to what extent a language designer can define its own activity diagrams to design specific documentations.

Acknowledgements

We thank the ThingML language designers [17] for their help on ThingML and on the evaluation. We thank Benoit Combemale for his insightful comments on this paper.

References

[1] M. Mernik, J. Heering, A. M. Sloane, When and How to Develop Domain-Specific Languages, ACM Computing Surveys (CSUR) 37 (4) (2005) 316–344.
URL `https://dl.acm.org/citation.cfm?id=1118892`

[2] B. Combemale, Towards Language-Oriented Modeling, Habilitation, Université de Rennes 1 (Dec. 2015).
URL `https://hal.inria.fr/tel-01238817/`

[3] J. Sprinkle, M. Mernik, J.-P. Tolvanen, D. Spinellis, What kinds of nails need a domain-specific hammer?, IEEE software 26 (4).

[4] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. Kats, E. Visser, G. Wachsmuth, DSL engineering: Designing, implementing and using domain-specific languages, CreateSpace, 2013.

[5] L. Bettini, Implementing Domain-Specific Languages with Xtext and Xtend, Packt Publishing Ltd, 2013.

[6] T. Degueule, B. Combemale, A. Blouin, O. Barais, J.-M. Jézéquel, Melange: A Meta-language for Modular and Reusable Development of DSLs, in: In Proc. of SLE'15, 2015.
URL `https://hal.inria.fr/hal-01197038`

[7] J. Cánovas, J. Cabot, Enabling the Collaborative Definition of DSMLs, in: International Conference on Advanced Information Systems Engineering, Valence, Spain, 2013.
URL `https://hal.inria.fr/hal-00818943`

[8] M. Fowler, Language workbenches: The killer-app for domain specific languages.

[9] M. Bravenboer, E. Visser, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 365–383.

[10] G. Uddin, M. P. Robillard, How api documentation fails, IEEE Software 32 (4) (2015) 68–75.

[11] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, Kompren: modeling and generating model slicers, Software & Systems Modeling 14 (1) (2015) 321–337.
URL `https://hal.inria.fr/hal-00746566v2`

[12] A. Blouin, B. Combemale, B. Baudry, O. Beaudoux, Modeling Model Slicers, in: ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems, 2011, pp. 62–76.
URL `https://hal.inria.fr/inria-00609072`

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

[14] J.-M. Favre, D. Gasevic, R. Lämmel, E. Pek, Empirical language analysis in software linguistics, in: International Conference on Software Language Engineering, Springer, 2010, pp. 316–326.

[15] A. Van Deursen, P. Klint, Domain-specific language design requires feature descriptions, CIT. Journal of computing and information technology 10 (1) (2002) 1–17.

[16] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering (ICSE'81), IEEE Press, 1981, pp. 439–449.

[17] N. Harrand, F. Fleurey, B. Morin, K. E. Husa, Thingml: a language and code generation framework for heterogeneous targets, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ACM, 2016, pp. 125–135.

[18] D. J. Sheskin, Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition, Chapman & Hall/CRC, 2007.

[19] J. García, O. Diaz, M. Azanza, Model transformation co-evolution: A semi-automatic approach, in: International Conference on Software Language Engineering, Springer, 2012, pp. 144–163.

[20] A. Cicchetti, D. Di Ruscio, R. Eramo, A. Pierantonio, Automating co-evolution in model-driven engineering, in: Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE, IEEE, 2008, pp. 222–231.

[21] A. Blouin, N. Moha, B. Baudry, H. Sahraoui, J.-M. Jézéquel, Assessing the Use of Slicing-based Visualizing Techniques on the Understanding of Large Metamodels, Information and Software Technology 62 (0) (2015) 124 – 142. `doi:10.1016/j.infsof.2015.02.007`.
URL `https://hal.inria.fr/hal-01120558`

[22] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, T. Ratchford, Automated API property inference techniques, IEEE Transactions on Software Engineering 39 (5) (2013) 613–637. `doi:10.1109/TSE.2012.63`.

[23] E. Arisholm, L. C. Briand, S. E. Hove, Y. Labiche, The impact of uml documentation on software maintenance: An experimental evaluation, IEEE Transactions on Software Engineering 32 (6) (2006) 365–381.

[24] D. Sadigh, S. A. Seshia, M. Gupta, Automating exercise generation: A step towards meeting the MOOC challenge for embedded systems, in: Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education, ACM, 2012, p. 2.
URL `https://dl.acm.org/citation.cfm?id=2530546`

[25] O. Polozov, E. O'Rourke, A. M. Smith, L. Zettlemoyer, S. Gulwani, Z. Popovic, Personalized mathematical word problem generation., in: IJCAI, 2015, pp. 381–388.

[26] P. Gómez-Abajo, E. Guerra, J. de Lara, A domain-specific language for model mutation and its application to the automated generation of exercises, Computer Languages, Systems & Structures.
URL `http://dx.doi.org/10.1016/j.cl.2016.11.001`

[27] L. Ab. Rahim, J. Whittle, A survey of approaches for verifying model transformations, Software & Systems Modeling 14 (2) (2015) 1003–1028.
URL `http://dx.doi.org/10.1007/s10270-013-0358-0`

[28] F. Fleurey, B. Baudry, P.-A. Muller, Y. L. Traon, Qualifying input test data for model transformations, Software & Systems Modeling 8 (2) (2009) 185–203.
URL `https://hal.inria.fr/hal-00880639/`

[29] S. Sen, B. Baudry, J.-M. Mottu, Automatic model generation strategies for model transformation testing, in: International Conference on Theory and Practice of Model Transformations, Springer, 2009, pp. 148–164.
URL `https://hal.inria.fr/hal-00461267/`

[30] F. Fleurey, B. Baudry, P.-A. Muller, Y. Le Traon, Towards dependable model transformations: Qualifying input test data, Journal of Software and Systems Modeling (SoSyM).
URL `https://hal.inria.fr/inria-00477567/`

[31] M. Gogolla, A. Vallecillo, L. Burgueno, F. Hilken, Employing classifying terms for testing model transformations, in: Model Driven Engineering Languages and Systems (MODELS), 2015 ACM/IEEE 18th International Conference on, IEEE, 2015, pp. 312–321.
URL `https://doi.org/10.1109/MODELS.2015.7338262`

[32] E. Batot, H. Sahraoui, A generic framework for model-set selection for the unification of testing and learning MDE tasks, in: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, ACM, 2016, pp. 374–384.
URL `https://dl.acm.org/citation.cfm?id=2976785`

[33] T. Pati, S. Kolli, J. H. Hill, Proactive modeling: a new model intelligence technique, Software & Systems Modeling 16 (2) (2017) 499–521. `doi:10.1007/s10270-015-0465-1`.
URL `http://dx.doi.org/10.1007/s10270-015-0465-1`

[34] I. Jacobson, Object-oriented development in an industrial environment, in: ACM SIGPLAN Notices, Vol. 22, ACM, 1987, pp. 183–191.

[35] T. Yue, L. C. Briand, Y. Labiche, A use case modeling approach to facilitate the transition towards analysis models: Concepts and empirical evaluation, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2009, pp. 484–498.