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]*Univ Rennes, INSA Rennes, Inria, CNRS, IRISA – France*

Abstract

Domain-specific languages (DSL) are developed for answering specific problems by leveraging the expertise of domain stakeholders. The development of DSLs requires a significant software engineering effort: editors, code generators, *etc.*, must be developed to make a DSL usable. Documenting a DSL is also a major and time-consuming task required to promote it and address its learning curve. Recent research work in software language engineering focus on easing the development of DSLs. This work focuses on easing the production of documentation of textual DSLs. The API documentation domain identified challenges we adapted to DSL documentation. Based on these challenges we propose a model-driven approach that relies on DSL artifacts to extract information required to build documentation. Our implementation, called *Docywood*, targets two platforms: *Markdown* documentation for static web sites and Xtext code fragments for live documentation while modeling. We used *Docywood* on two DSLs, namely *ThingML* and *Target Platform Definition*. Feedback from end users and language designers exhibits qualitative benefits of the proposal with regard to the DSL documentation challenges. End user experiments conducted on *ThingML* and *Target Platform Definition* show benefits on the correctness of the created models when using *Docywood* on *ThingML*.

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

1. Introduction

According to Mernik *et al.*, domain-specific language (DSLs) are software languages that "*provide notations and constructs tailored toward a particular application 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 and compilers are examples of core components that compose a DSL ecosystem. Current research efforts focus on easing the development of specific parts of DSLs to reduce their development cost and maintenance [5, 6, 7, 8].

The work proposed in this paper aims at opening new research questions and proposing new tools for DSL maintenance 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*" [9]. Documenting a DSL is indeed another major 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*": languages are hard to learn and the use of many languages will be much more complicated than using a single language [9]. By studying parallels made between DSLs and APIs (*Application Programming Interface*) [1, 10, 11], we identified four technical properties that end-user DSL documentation tools should possess: 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 task that can lead to documentation obsoleteness; 4) Providing code examples to illustrate each concept of a DSL is a time-consuming task. Documentation is known as improving the usability of the documented artifact [12]. So, these four properties aim at answering a more general challenge that is improving the usability of DSLs by improving their end user documentation.

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 [13, 14] to

keep their elements that focus on this concept. A piece of documentation, dedicated to this concept, is then produced and is composed of: an illustrative example; explanations about the concept and its possible parameters, in natural language. 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) as *Java* code to be seamlessly integrated in the Xtext [5] DSL editor and then to provide DSL users with live documentation by content assist. 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) [15] and Xtext. We validated the proposal through an experiment that involves: 17 subjects; two third-part modeling DSLs, namely *ThingML*[2] [16] and *Target Platform Definition*,[3] designed for a computer science audience; two language designers of *ThingML*. The results of the experiment exhibit qualitative benefits of the proposal with regard to the five DSL documentation challenges. Both subjects and language designers identified several possible improvements. The quantitative results exhibit benefits regarding the correctness of the created models when using the generated documentation in addition to the official one for ThingML.

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 details the evaluation of the approach. Section 5 discusses related work. Section 6 concludes the paper and gives insights for future work.

2. Problem Statement

First, we introduce an illustrative example use throughout this paper to illustrate the approach. 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. 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 is 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.

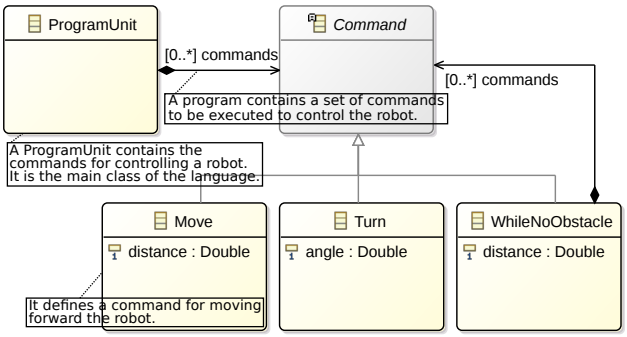


Figure 1: The metamodel of the *Robot* language

[1] See: `https://github.com/arnobl/comlanDocywood`
[2] See `http://thingml.org/`
[3] See `https://github.com/mbarbero/fr.obeo.releng.targetplatform`

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 parameters 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. Overall Objectives

Provide a support to automate the production of DSL documentation.

2.2.1. Technical properties

Software languages are software too [17] and relations between APIs (*Application Programming Interface*) and DSLs have been established [1, 10]. So, the relationship can be drawn between DSL and API documentation to precise the overall objectives. API documentation pitfalls have been identified from end users feedback [11]. From these pitfalls we derived four technical properties that end user documentation tools should possess.

Property #1 – Documentation coverage guarantee. Incomplete documentation has been identified as the most important problem that affects API documentation. This property aims at assuring that documentation coverage criterion has been considered. 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. For example with the illustrating example, all the concepts of the *Robot* metamodel must be documented: classes such as *ProgramUnit*, attributes such as *angle*, and relations such as *commands*.

Property #2 – Documentation contextualization. Documentation has to be contextualized according to the current needs of the language users. This may limit the *bloat* and *tangled information* issues brought by providing large and not fully relevant chunks of texts to users. For example with the illustrating example, when an end user of the *Robot* language works on the *Turn* class, only the documentation that concerns this class and related elements (*e.g.,* its mandatory attribute angle) must provided.

Property #3 – Multi-platform documentation generation. Maintaining documentation manually over one or several platforms (several language workbenches, websites, *etc.*) is a complex task that can lead to documentation obsoleteness. For example with the illustrating example, documentation to be integrated within the Xtext editor or within wiki pages could be generated. In case of changes in the language artifacts (*e.g.,* the *Robot* metamodel or its grammar), the documentation must be easily updated.

Property #4 – Example-based documentation. While users generally appreciate examples in the documentation, these examples must be supplemented with adequate explanations. For example with the illustrating example, the documentation that concerns the class *Move* must have small yet relevant illustrative examples, such as: `move(75)`.

In the early stages of the DSL development process, domain analysis can help in identifying the concepts of the DSL and in producing some general documentation [18]. Downstream of the development process, language designers can manually write documentation using common tools such as *EcoreTools* [15]. These different approaches do not possess the four properties that characterize the scientific problem of easing the production of DSL documentation: if language designers can document a metamodel, this documentation is mainly dedicated to other language designers to build the DSL ecosystem. The four properties concern the documentation dedicated to *DSL users*, *i.e.,* to users that will use the DSL through its ecosystem (*e.g.,* editors). Moreover, metamodel documentation does not usually 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.

2.2.2. Usefulness of the approach

The four aforementioned properties focus on features tools that produce DSL documentation should support. From an end user perspective, these four properties aim at achieving a more general challenge, defined as follows.

Challenge – Improving DSL usability with end user documentation. Documentation is known as improving the usability of the documented artifact [12]. A tool that possesses the four aforementioned properties must improve the usability of DSLs by improving their end user documentation. Based on the four aforementioned properties, this paper proposes a new DSL documentation approach to automate the production of DSL documentation. The evaluation of the approach focuses on discussing the global challenge by measuring the impact of the documentation generated by the implementation of the approach on the usability of two industrial DSLs.

3. Approach

This section presents the approach in details. 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 3.5.

3.1. Documentation Content and Structure

The output documentation produced by the approach are pieces of documentation that document each concept of the DSL under study.

A piece of 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 illustrative example; references to other related pieces of documentation.

The illustrative example of a piece of 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 can still be applied but no explanation about the concepts is provided.

Figure 2 depicts one piece of documentation of the *Robot* DSL. The piece of 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 documentation explains the different elements not already explained in other pieces of documentation: we assume that *ProgramUnit*has its own piece of documentation so *ProgramUnit*is not explained in the documentation related to *Move*. The generated text uses: the documentation extracted from the metamodel of Figure 1; the concrete syntax grammar of Listing 2; an illustrative model (Listing 1). Links to the documentation of related concepts are provided. For example, the documentation related to *Move* ends with a link to the definition of a *ProgramUnit*.

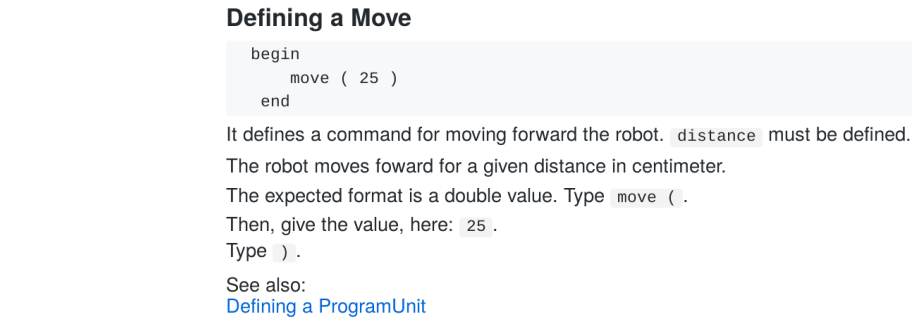


Figure 2: The piece of documentation that explains the *Move* concept of the *Robot* DSL

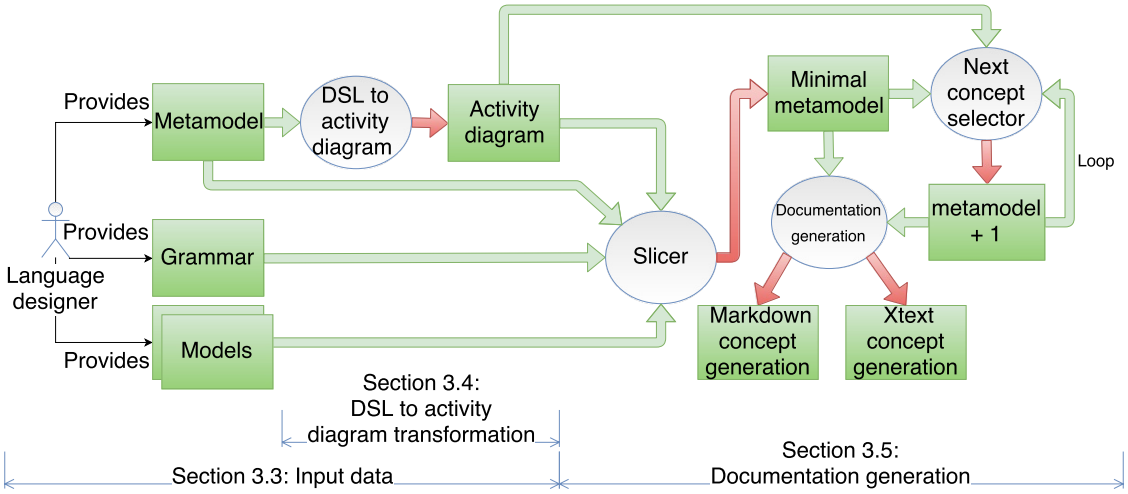


Figure 3: The different steps of the approach

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 3 depicts the different steps of the approach. The language designer has to provide as input data to the approach the documented metamodel of a DSL, its grammar, and models that will serve as examples in the documentation.

An activity diagram can also be provided as input data to the approach. This activity diagram orders the generation of the pieces of documentation.

This diagram is automatically generated at the first run of the approach on a given DSL. Once generated, language designers can edit this diagram to: change the generation order of the pieces of documentation; merge, remove, or modify pieces of documentation. Section 3.3 details the characteristics of the input data.

Section 3.4 details the generation of the activity diagram (*DSL to activity diagram transformation* in Figure 3). The process then takes the first activity of the activity diagram to produce the root piece of documentation of the DSL. The root piece of 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 piece of documentation follows step 3.5 of Figure 3. First, the *Slicer* slices the metamodel to keep only its minimal mandatory elements. Model slicing is an operation that extracts a subset of model elements [13, 14]. Second, the *Documentation generation* module uses the sliced metamodel, the grammar, and the input models to generate the root piece of documentation. Then, the sliced metamodel is augmented with one concept from the original metamodel thanks to the *next concept selector* step of Figure 3. Model slicing is also used during this step to get the mandatory elements related to the concept newly added. The documentation generation and incre-

mentation ("*next concept selector*") steps repeat over the successive sliced metamodels until all the activities of the activity diagram are considered. Section 3.5 details the documentation generation.

3.2.2. Scope of the approach

First, the proposed approach detailed in this section focuses on external grammar-based and textual DSLs. External DSLs represent 50% of the DSLs [8]. 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. Third, the approach is generalizable to any meta-metamodel that supports the class, attribute, relation, inheritance, and cardinality concepts. This is the case of several widely used meta-metamodel languages, such as Ecore (used in the implementation), UML,[4] or MetaEdit+ [19]. Finally, we originally designed the approach for DSLs that target a computer science audience. The approach may be used on DSLs that target other audiences, but this requires supplementary experiments, that is out of the scope of the current paper.

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 a piece of documentation. If the metamodel is not documented the process still works but textual explanations will be absent from the generated documentation.

The grammar of a DSL is used to: give instructions in pieces of documentation on how to write the current concepts using the concrete syntax; produce code examples from the input models (see Section 3.5).

The input models must cover all the concepts of the DSL. They indeed constitute the examples on which documentation will be based. Their usages are detailed in Section 3.5.

3.4. DSL to activity diagram transformation

The first time a language designer runs the approach for a given DSL, a UML activity diagram is automatically produced. This activity diagram describes the sequences of the pieces of documentation. The goals of this activity diagram are twofold. First, it orders the generation of the pieces of documentation, as detailed in Section 3.4.1. Second, it allows the language designers to change this generation order, as detailed in Section 3.4.2.

3.4.1. Activity diagram generation

Algorithm 1 details the generation of the activity diagram. Figure 4 illustrates this algorithm using the *Robot* DSL. The first goal of the activity diagram is to order the generation of the pieces of documentation. This order is mandatory as a generated piece of documentation will contain details about its parent documentation. For example, the *Move* class of the *Robot* language should be explained after the *ProgramUnit* concept as there is a composition relation between these two classes in the *Robot* metamodel (see Figure 1).

The algorithm follows the structure of the metamodel. The first step consists in detecting the root class of the metamodel (line 2). For simplicity, this description and Algorithm 1 make the assumption that a metamodel has a single root class. The approach, however, supports several root classes. In the case of the *Robot* metamodel, the root class is *ProgramUnit*(Figure 4a). The activity corresponding to the root class is then produced (line 5). Figure 4b shows the created activity "*Defining a ProgramUnit*". 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 corresponding documentation (see *activity.getData()*, line 6 in Algorithm 2, in Section 3.5.1).

A crucial step in the approach is the use of model slicing [14, 13] to get the minimal metamodel chunk from a set of slicing criteria, *i.e.,* a set of metamodel elements. Model slicing is a model comprehension technique inspired by program slicing [20]. The process of model slicing involves *extracting* from an input model a subset of model elements

[4] `http://www.omg.org/spec/UML`

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, ∅, activityDiag)
6:      return activityDiag
7:
8:  procedure genActivity(elt, metamodel, prevActivity, explained, diag)
9:      activity := createActivity(elt)
10:     explained := explained ∪ sliceMetamodel(metamodel, elt)
11:     diag := diag ∪ {prevActivity → activity}
12:     elements := getElements(elt) \ explained
13:     for each e ∈ elements do
14:         genActivity(e, metamodel, activity, clone(explained), diag)
```

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 criterion is currently the *ProgramUnit*class. To get the minimal metamodel chunk from the *ProgramUnit*class, all the metamodel elements strongly connected to this class are included (line 10): all the attributes and relations that have a lower cardinality greater than 0 are recursively included in the metamodel chunk. The elements of the sliced metamodel will be explained in the same piece of documentation (if not already explained in another piece of documentation).[5] The *ProgramUnit*class does not have any mandatory element so that the minimal metamodel chunk is composed of the *ProgramUnit*class only (Figure 4a).

Then, all the elements contained by the current element *elt* (*e.g.,* a class has attributes and references, a reference has a target class) and not already explained in generated pieces of documentation are gathered (line 12). For each of these elements, an activity will be recursively produced following the current activity (lines 13 and 14).

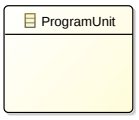
For example, Figure 4c is the sliced *Robot* metamodel produced for the second round of the activity diagram generation. The slicing criteria are now the class *ProgramUnit*plus its single element: the composition *commands* that targets the abstract class *Command*. Because *Command* is abstract, its sub-classes are directly considered (sub-classes are also considered for concrete classes). The first sub-classes of *Command* is *Move*. *Move* has a mandatory attribute *distance*. From this new sliced metamodel, a new activity "*Defining a* Move" is produced, as depicted by Figure 4d. This activity will document both the class *Move* and its attribute *distance*.

Figure 4e is the next sliced *Robot* metamodel where another sub-class of *Command* is used, namely the class *Turn*. This class has a mandatory attribute *angle* that is included in the sliced metamodel. The activity diagram is then completed with another activity "*Defining a* Turn" to document both the class *Turn* and its attribute *angle*. Note that this new activity is not placed following the activity related to *Move* produced during the previous round: there is no relation from *Move* to *Turn*.

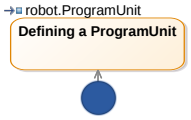
Figure 4g is the next sliced *Robot* metamodel where the last sub-class of *Command* is used: *WhileNoObstacle*. This class has a mandatory attribute *distance* included in the sliced metamodel. A new activity "*Defining a* WhileNoObstacle" is created to document both the class *WhileNoObstacle* and its attribute *distance*. Note that sliced metamodel of this round does not include the composition *commands* of *WhileNoObstacle*: this composition is not mandatory and thus not included by default in the metamodel. This explains why the new activity created for *WhileNoObstacle* does not refer to this composition (see Figure 4g). Note that this new activity will not explain the target class *Command*, or its sub-classes, of this composition since activities already exist for these classes. All the metamodel elements encountered are registered (line 10) and then discarded from the set of elements to be covered by the activity currently produced. If all elements have been traversed, this last set is empty and then the recursive call ends (lines 13 to 14).

The last round of the process on the Robot DSL focuses on the composition *commands* of *WhileNoObstacle* that

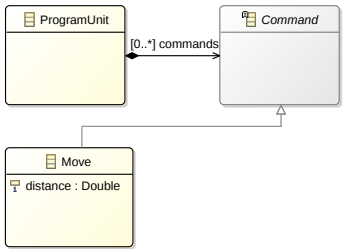
[5]The fact that the attributes with a lower cardinality of 0 are not explained in the same documentation is a design choice. We prefer to provide the most simple documentation possible before providing the details.



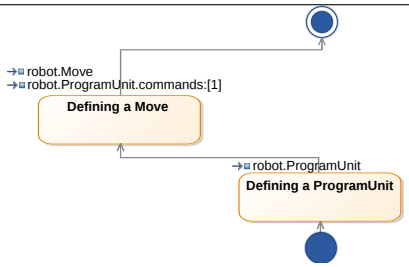
(a) The sliced metamodel of *Robot* using *ProgramUnit*as slicing criterion



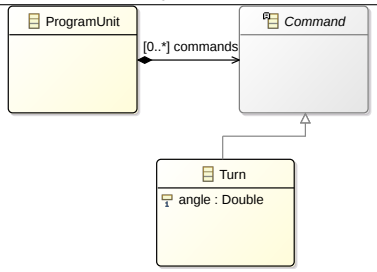
(b) The activity produced from the metamodel sliced using *ProgramUnit*



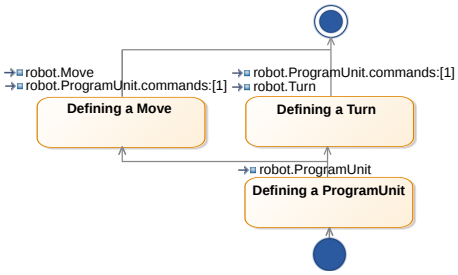
(c) The sliced metamodel of *Robot* using *ProgramUnit*, *commands*, and *Move* as slicing criteria



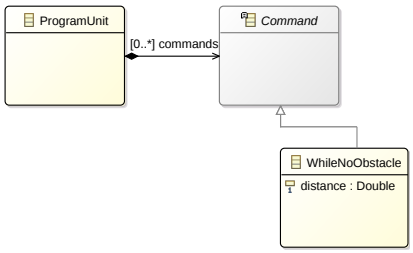
(d) The activity produced from the sliced metamodel on the left where *Move* has its own activity



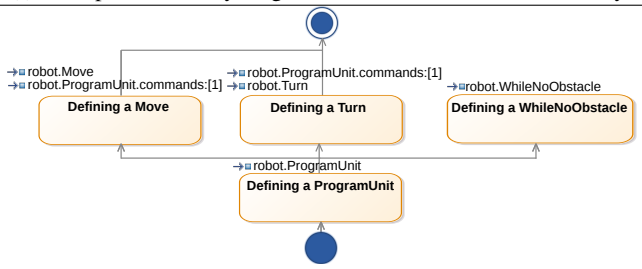
(e) The sliced metamodel of *Robot* using *ProgramUnit*, *commands*, and *Turn* as slicing criteria



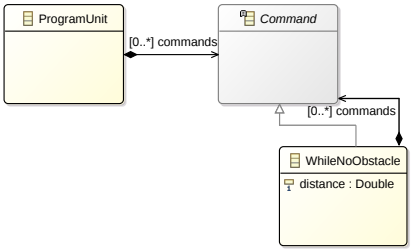
(f) The updated activity diagram where *Turn* has its own activity



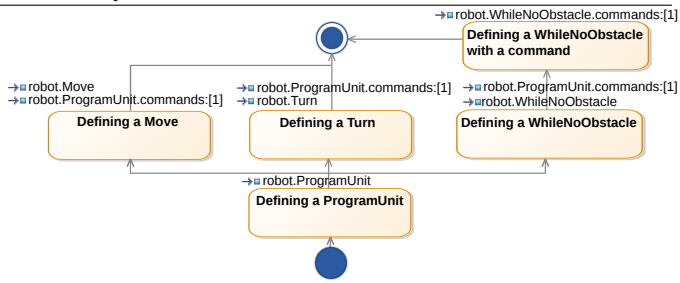
(g) The sliced metamodel of *Robot* using *ProgramUnit*, *commands*, and *WhileNoObstacle* as slicing criteria



(h) The updated activity diagram where *WhileNoObstacle* has its own activity



(i) The sliced metamodel of *Robot* using *ProgramUnit*, *commands*, *WhileNoObstacle* and its *commands* composition



(j) The updated activity diagram where an activity that focuses on the composition *commands* of *WhileNoObstacle* has been created

Figure 4: Illustration of Algorithm 1 using the *Robot* example

is now included in the sliced metamodel (see Figure 4i). The final activity diagram is depicted by Figure 4j. It now contains a last activity that focuses on the composition *commands* of the class *WhileNoObstacle*.

The algorithm follows coverage criteria to ensure that all the elements of a DSL are explained in one piece of documentation: all the attributes, references, and classes of the DSL metamodel are covered by at least one piece of documentation. Algorithm 1 ensures these coverage criteria by recursively producing activities.

3.4.2. Activity diagram usage

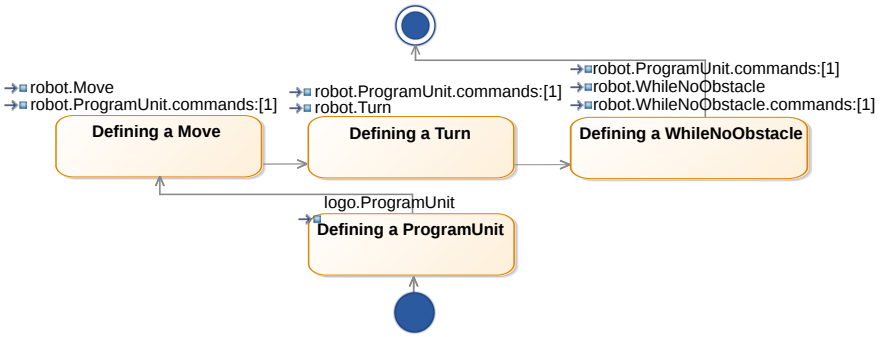


Figure 5: The UML activity diagram of Figure 4j modified by a language designer

Once the activity diagram has been generated, a language designer can merge, remove, modify, and change the order of the activities. The goal is to let language designers customize the documentation 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 4j we modified. The activity dedicated to *Turn* is now an output activity of the *Move* activity. The documentation related to *Turn* generated from this modified activity diagram (see Figure 6b) now contains a *Move* command in its code example and a reference to the documentation related to *Move*, contrary to the documentation related to *Turn* generated from the default activity diagram (see Figure 6a). Note that the *Move* command is not explained in the piece of documentation of Figure 6b but in its own one.

A second modification brought to the original activity diagram is the merge of the activities "*Defining a* WhileNoObstacle" and "*Defining a* WhileNoObstacle *with a command*". The goal of this change is to explain in the same piece of documentation, the *WhileNoObstacle* class and its optional reference *commands*.

Defining a Turn

```
begin
    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

(a) A piece of documentation that explains the *Turn* concept of the *Robot* DSL. This documentation is generated from the default activity diagram depicted by Figure 4j.

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
Defining a Move

(b) A piece of documentation that explains the *Turn* concept of the *Robot* DSL. This documentation is generated from the manually modified activity diagram depicted by Figure 5.

Figure 6: Editing the activity diagram may have impacts on the generated documentation.

By default, the generated activity diagrams respect the coverage criteria detailed in Section 2.2. A language

designer may also define her own activity diagrams to design specific documentations, such as tutorials. This point, however, goes beyond the scope of this paper.

3.5. Documentation generation

This section details the documentation generation process.

3.5.1. General process

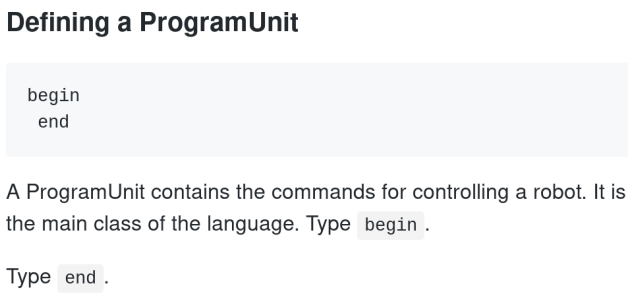


Figure 7: A piece of documentation that explains the class *ProgramUnit*of the *Robot* DSL

The documentation generation process takes as input the metamodel, the grammar, a set of models, and the UML activity diagram. Algorithm 2 details the generation process. Figures 6 and 7 illustrate the generated documentation using the *Robot* example. The generation is driven by the activity diagram (*cf.* Section 3.4 for the automatic production of the activity diagram at the first run of the approach): one activity is used to generate one piece of documentation. The documentation generation 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 this activity (*i.e.,* the class *ProgramUnit*) are then fetched (line 6). Using these metamodel elements, a model slicer slices the metamodel to get the minimal metamodel that uses the elements of the current activity (line 8). A second model slicer is then used to slice a model using this minimal metamodel (line 9). The goal of this second slicing is to get a model that only contains the elements contained in the sliced metamodel. The model will be used to explain the current concept. For example with *ProgramUnit*, the input model is given in Listing 1. The sliced model that only uses the *ProgramUnit* class is depicted by Figure 7. In this sliced model, only the tokens (*begin*, *end*) that correspond to *ProgramUnit* are kept. To do so, the *Robot* grammar is also sliced to keep the necessary rule (line 16):
`ProgramUnit : 'begin' 'end';`

The text of a piece of documentation is then produced using this sliced metamodel and the sliced grammar (line 10). The grammar rules are analyzed to progressively provide both typing instructions and the text extracted from the metamodel documentation. For example with Figure 7, the first line "*A* ProgramUnit *[...] the language*" comes from the metamodel documentation. The typing instructions, "*Type* begin. *Type* end." are generated from the sliced grammar.

Finally, all the output activities of the current one are recursively treated (lines 12 and 13). For example, the activity "*Defining a* Turn" follows the activity "*Defining a* ProgramUnit". The generated documentation that explains *Turn* is depicted by Figure 6a. The input model is now sliced with both the *ProgramUnit* and *Turn* classes. The slicer produces a model that contains one instance of both concept. A command *turn(-100)* is now contained into the *begin/end* block. The end of this piece of documentation contains a reference to the previous activities, here to "*Defining a* ProgramUnit".

3.5.2. Markdown and Xtext generation

The Markdown generation of a given piece of documentation is summarized by the lines 15 to 18 of Algorithm 2. The model that will illustrate a piece of 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 analyzed to extract the sequence of typing actions that explain how to write the model

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)
```

(line 17). For example, given the *Robot* grammar (Listing 2) and the root activity "*Defining a ProgramUnit*", Figure 8 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 4a) the *Command* class that corresponds to the second action does not exist. This action is thus ignored. So, the resulting piece of documentation is composed of the actions "*begin*" and "*end*" only. The resulting *Markdown* documentation is shown in Figure 7. The two first sentences of the text come from the metamodel documentation.

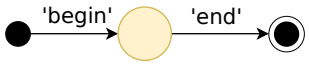


Figure 8: The sequence of generated actions for the activity "*Defining a ProgramUnit*" (see Figure 4b)

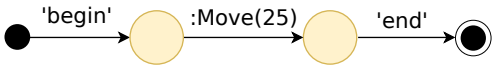


Figure 9: The sequence of generated actions for the activity "*Defining a Move*" (see Figure 4d) and the example model of Listing 1

Given the second activity "*Defining a Move*" and the model of Listing 1, Figure 9 depicts the sequence of actions. The same reasoning applies. Contrary to the previous example, the *Command* class exists in the sliced metamodel (see Figure 4c). 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 this piece of documentation: *move(25)*. Figure 2 shows the output *Markdown* piece of 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 than for "*Defining a Move*".

By default, the process uses a metamodel element one time in a piece of documentation, *e.g.,* the documentation dedicated to Figure 2 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 4j). 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 10. The

Xtext fragment generation integrates the generated Markdown documentation into the Xtext API.

4. Evaluation

The evaluation of the proposed approach focuses on its correctness and its usefulness. The correctness of the approach is evaluated by ensuring that the required properties mentioned in Section 2.2.1 are considered (Section 4.1). Since the documentation is supposed to help in the usability of the documented object, usefulness of the approach is evaluated regarding the provided increase of the DSL usability (Section 4.2). The materials of the evaluation are available on the companion web page of this paper.[6]

4.1. Analytical validation of the tooling

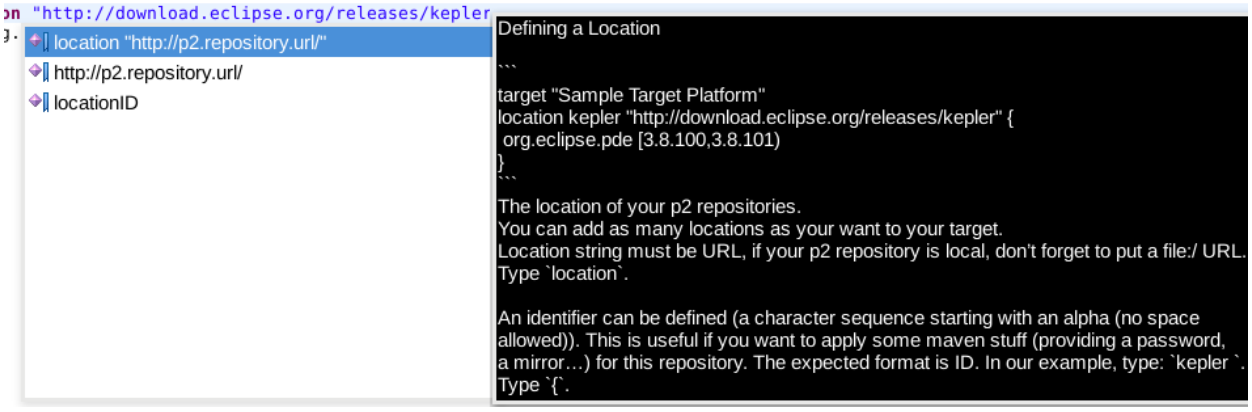


Figure 10: Example of contextualized documentation in an Xtext editor

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) [15]. *Docywood* also relies on Xtext [5] to manage the description of the DSL concrete textual syntaxes. The model slicing technique used in the approach is performed by Kompren, a DSL for defining model slicers [13, 14]. *Docywood* takes as input: an Ecore model (the metamodel of the DSL); instances of this metamodel (models of the DSL); the Xtext model that defines the concrete textual syntax of the DSL. *Docywood* can also take as input the UML activity diagram that describes the generation order of the pieces of documentation. When not provided as input, this diagram is automatically generated during the execution of *Docywood*. Language designers can then edit it for future documentation generations.

We now discuss the four technical properties that a DSL documentation approach must possess. The proposed approach relies on coverage criteria to overcome the documentation coverage issue (property #1). The coverage criteria are based on the traversal strategies of the DSL metamodel. The approach is based on Ecore metamodels so that the traversal strategies use the relations (*i.e.,* references and compositions) to cover classes, attributes, and relations. The class, attribute, relation are supported by most of the current used meta-metamodel languages, such as UML and MetaEdit+. The current implementation *Docywood* has two limits in terms of documentation coverage. First, *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 has dedicated documentation. This is a technical choice made for *Docywood*, which can be changed in future versions. Second, metamodels may be supplemented with constraints declared using the OCL language [7] or within the type checker of an editor.

Regarding the property #2 – Documentation contextualization – *Docywood* enables user context adaptation of the documentation through two features. First, the generation process generates pieces of documentation. Each piece of

[6] See: `https://github.com/arnobl/comlanDocywood`
[7] `http://www.omg.org/spec/OCL/`

documentation focuses a specific concept of the DSL. The Xtext code generated by *Docywood* selects the appropriate piece of documentation for the content assist service. The Xtext generated code has a limitation: it strongly depends on the content assist service of the DSL editor, relying on the concrete syntax and the typing information. Consequently, the granularity of the contextualization is bound to model elements and their corresponding types. If the content assist service of the DSL editor has been already edited by language designers, the generated Xtext code fragment and the current content assist service must be merged.

Regarding the property #3 – Multi-platform documentation generation – a current issue is that language designers have to maintain documentation over different platforms (*e.g.,* on the web site of the DSL, embedded in tools). The documentation generation process is platform-independent. As a demonstration, the current implementation targets two platforms. First, Markdown documents are generated to be integrated on web sites as static documentation. Second, Xtext code fragments are generated to be seamlessly integrated in the code completion of the Xtext editor of the DSL and provide live documentation. We identified on limit in the proposed approach regarding documentation updates. 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 may be lost.

Regarding the property #4 – Example-based documentation – each piece of documentation has an illustrative example. This model is specifically customized to match the DSL concept of its piece of documentation. To do so, model slicing is used to slice input models provided by language designers. This feature has a limitation: the input models are provided by the language designer, so that they must be carefully selected to illustrate the DSL. Illustrative examples are provided with no context regarding their original model. So, they should be simple or explicit enough to be understood by end users.

4.2. Experimental validation

In this section we evaluate qualitatively and quantitatively [21] the benefits of the proposal by conducting an empirical experiment on real third-part DSLs. We first then use the GQM (*Goal Question Metric*) approach [22] to specify our experiment. We then describe the experimental protocol and detail the results of the experiment. Discussions conclude the evaluation.

4.2.1. Goals, Questions, and Metrics of the evaluation

Goal: to empirically observe whether the proposed approach overcomes the global DSL documentation challenge detailed in Section 2.2: evaluate whether the generated DSL documentation is useful for language designers and end users.

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?

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

Metrics: The metrics related to the quantitative RQ1 and RQ2 are described in the paragraph *Dependent variables* of Section 4.2.2. RQ3 and RQ4 are qualitative RQs based on discussions with the subjects and interviewed language designers. qualitative RQs "help to answer questions that involve variables that are difficult to quantify (particularly human characteristics such as motivation, perception, and experience)" [21]. RQ1 and RQ2 are qualitative RQs because of the difficulty to quantify the feelings of the subjects regarding the proposed approach.

Recent work focus on proposing metrics for DSL quality assessment [23, 24, 25, 26, 27, 28]. These metrics are designed to evaluate DSLs, but can be adapted to evaluate other types of software such as *Docywood* or the documentation it generates. As detailed in this section, we adapted the time and correctness metrics as well as qualitative questionnaire to our approach.

4.2.2. Experimental protocol

The *objects* of the experiment are two DSLs. The first DSL is ThingML,[8] a DSL for modeling embedded and distributed systems [16]. The second DSL is TPD (*Target Platform Definition*) [9] 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[10]; 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) [13, 14]. 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[11] and on the companion web page of this paper.

Tasks. We designed two exercises, one for ThingML and one for TPD. We measure the complexity of these exercises using the metrics *Abstraction* that counts the number of abstractions the subjects had to handle to do each exercise. The ThingML exercise consists in modeling a component that receives random values from a port and then 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 *Abstraction* value of the ThingML exercise is 55 model elements.

The TPD exercise consists in modeling a target platform that imports from different other sources specific plug-ins with specific versions for a given platform. The *Abstraction* value of the TPD exercise is 20 model elements.

Subjects. 17 subjects did the experiment: one master student, seven PhD students, four software engineers, two post-doc, and three researchers. We selected subjects in our research laboratory (around 700 persons) with the following criteria: they must have a background in software engineering and must be recurrent users of different DSLs.

Table 1: Exercises distribution among the 17 subjects

Subjects ID	Execution Order		Docywood usage		Knowledge in Eclipse	TPD	Distibuted system	ThingML
#1	TPD	thingml	Yes	No	2	1	3	1
#2	thingml	TPD	Yes	No	1	1	1	1
#3	TPD	thingml	No	Yes	1	1	1	1
#4	thingml	TPD	No	Yes	3	1	4	1
#5	TPD	thingml	Yes	No	1	1	1	1
#6	thingml	TPD	Yes	No	3	1	3	1
#7	TPD	thingml	No	Yes	4	1	2	1
#8	thingml	TPD	No	Yes	2	1	1	1
#9	TPD	thingml	Yes	No	4	1	2	1
#10	thingml	TPD	Yes	No	1	1	3	1
#11	TPD	thingml	No	Yes	2	1	1	1
#12	thingml	TPD	No	Yes	2	1	2	1
#13	TPD	thingml	Yes	No	1	1	1	1
#14	thingml	TPD	Yes	No	5	1	2	3
#15	TPD	thingml	No	Yes	2	1	2	3
#16	thingml	TPD	No	Yes	3	1	4	2
#17	thingml	TPD	Yes	No	1	1	1	1

Regarding RQ4, three language designers of each DSL were contacted by email.

[8] `http://thingml.org/`

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

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

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

Procedure. The experiment was tested by one subject (not counted in the 17 subjects) to check the tools and the protocol. The results of this subject were not considered. The experiment is based on a within-subjects design [29]: each subject did the experiment on both ThingML and TPD. Interference between the use of ThingML and TPD may result due to their execution order. To mitigate this classical threat of using a within-subjects design, we use counterbalancing: we designed a Latin squares table, as illustrated by Table 1, to vary the execution parameters. Half of the subjects did the ThingML exercise (resp. TPD exercise) using the official documentation of the DSL only ("*Docywood usage*": "*No*"). These subjects did the TPD exercise (resp. ThingML exercise) using the official documentation plus the documentation generated by *Docywood* ("*Docywood usage*": "*Yes*"). The goal is not to state whether the generated documentation is better than the official one or not, but whether it improves the modeling tasks of DSL users.

The experiment was conducted during two days in a dedicated and isolated room. The same supervisor, author of the paper, conducted the experiment. A desktop computer was provided. This computer had a 24 inch screen and all the materials for the experiment were already installed on a Linux system. Two versions of each DSL editor were installed: one with our generated Docywoord Xtext code fragment integrated and another one without it. Eclipse 4.6 was used with the following plug-ins: Xtext 2.11.0; EMF 2.12.0. Subjects came randomly one by one in the room. We asked the subjects not to talk each other about the experiment. Their skills on ThingML and TPD were asked just before the experiment to choose a slot in Table 1: we aimed at having the same number of subjects with similar knowledge on a given DSL that used this DSL with and without Docywood. To do so, each subject had to fill a form with the following four questions:
"*Knowledge in embedded and distributed systems (select the corresponding value, 1 = no knowledge, 5 = expert)*";
"*Knowledge in the ThingML DSL*".
"*Knowledge in Target Platform (Eclipse plugins / platform)*".
"*Knowledge in the DSL Target Platform Definition*".
The PDF version of the form and the resulting subjects data are available on the companion web page. Table 1 reports these data for each subject. Once filled, we selected a slot in Table 1. The supervisor then explained the first exercise and how to use the documentations. The supervisor launched: the software systems necessary for this exercise; a web browser on the official documentation of the current DSL and, if *Docywood usage = yes*, on its generated documentation we put on the companion web page of this paper. The subject read the text of the first exercise and was allowed to ask questions about it to the supervisor. The subject notified the supervisor when he/she started the exercise. Questions were not allowed during an exercise. The subject notified the supervisor when he/she ended the exercise. No time limit was used: a subject ended when he/she finished or withdrew from the exercise. The same process was used for the second exercise. The subjects had no way, but the type checker of the DSL editor, to test or check the models they created. Finally, the subjects had to fill a form with qualitative questions. The content of this form is described below in the results discussion of RQ3.

Dependent variables. To answer RQ1 and RQ2 we defined two measures frequently used in usability evaluations of DSLs and their related tools [28]:

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 TPD.

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 under study. In this email, we began by quickly presenting the Docywood project, and then we provided the produced artifacts for the addressed DSL (ThingML or TPD). Finally, we asked them three questions:

1. Do you think that DSL development can benefit from such an approach?
2. Is the produced documentation relevant?
3. Can you think of any benefit / drawback?

We also precised that they could add any extra comment.

4.2.3. 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 [30] (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). We use the Cohen's d index to measure the effect size between means [30].

Dependent variables	**Mean Official + *Docywood***	**Mean Official**	**Mean Diff**	**Cohen's *d***	**Significance p-value**
$CORRECT_{THINGML}$	92.9%	71.3%	**+21.6%**	**1.18**	**0.04216**
$CORRECT_{TPD}$	93.1%	93.9%	-0.8%	0.18	0.7588
$TIME_{THINGML}$	31.14m	39.33m	-8.19m	0.61	0.4807
$TIME_{TPD}$	11.34m	10.53m	41s	0.17	0.9626

Table 2: Results of *CORRECT* and *TIME* on ThingML and TPD with their official documentation compared to their official documentation plus the one generated by *Docywood*.

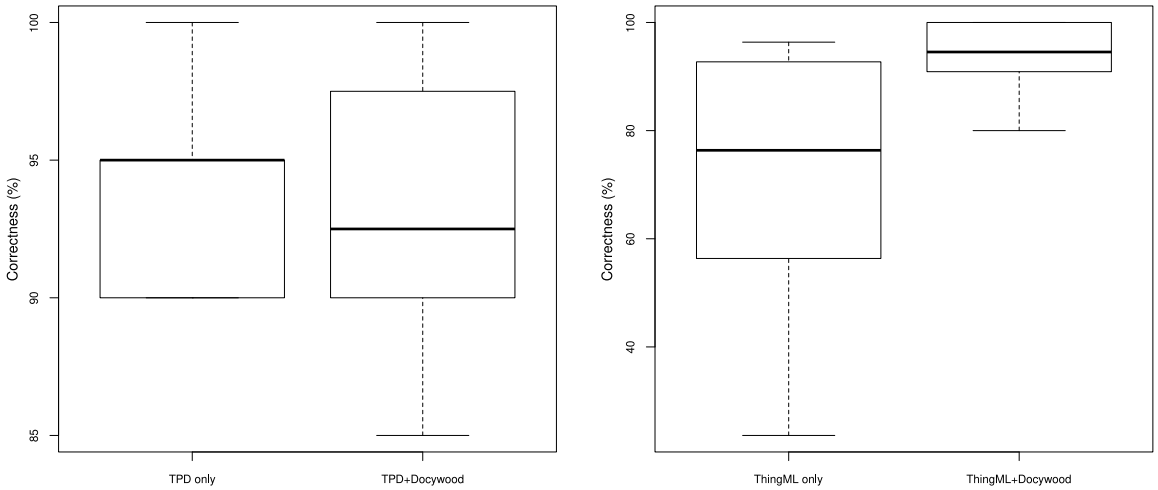


Figure 11: Box plots of *CORRECT* on ThingML and TPD with their official documentation compared to their official documentation plus the one generated by *Docywood*.

RQ1. Regarding the correctness of the models designed by the subjects (Table 2 and fig. 11), the average correctness measured shows a benefit when using the documentations generated by *Docywood* on ThingML (+21.6%). According to the Cohen's d test, this increase is large (1.18). This result 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 ThingML.

RQ2. Regarding the time spent by the subjects to perform the exercises (Table 2 and fig. 12), 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 textual and metamodel-based DSLs to do their first models faster. One possible explanation is that the generated documentation requires time to be read by the subjects. We also observed subjects that read all the documentation available before starting the exercises while other subjects iterate between the editor and the documentation. However, 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 work.

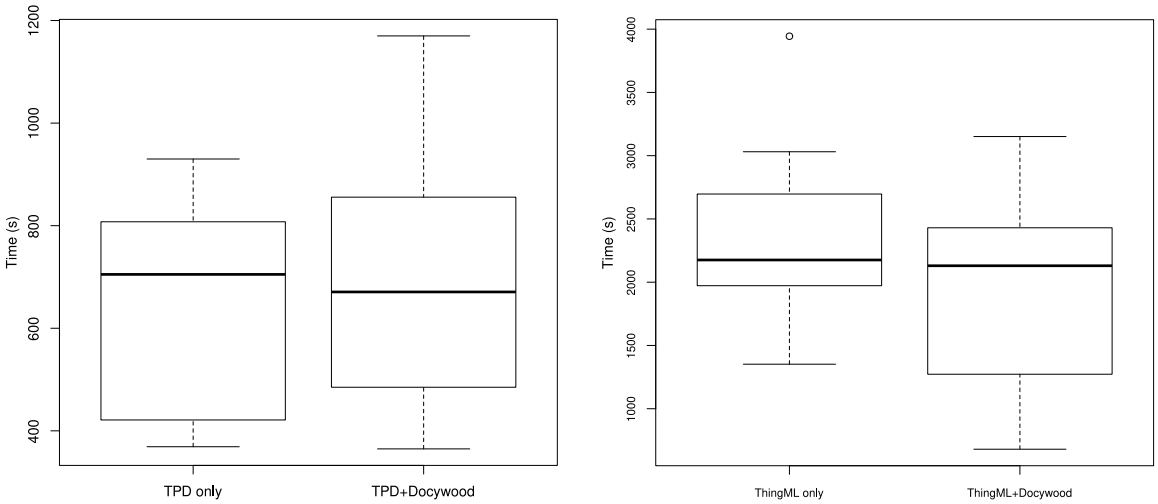


Figure 12: Box plots of *TIME* on ThingML and TPD with their official documentation compared to their official documentation plus the one generated by *Docywood*.

RQ3. Several points were anonymously evaluated (with a score on a scale of 1 to 5, 1 meaning useless, 5 meaning mandatory / very good) by the subjects at the end of their experiment:

- the global usefulness of the generated documentation (mean of 3.65);
- the usefulness of the code examples (mean of 3.88);
- the readability of the documentation (mean of 3.47);
- the completeness of the documentation (mean of 3.47);
- the usefulness of the coding instructions in the documentation (mean of 3).

The results are plotted in Figure 13. Subjects were then invited to give free and anonymous comments.

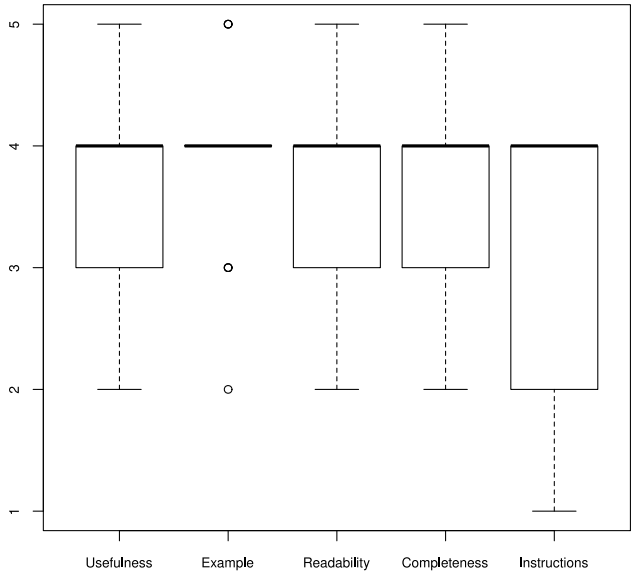


Figure 13: Anonymous evaluations of the generated documentation by the subjects

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 valued 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 been detailed: "*the generated documentation is not self-contained*"; "*I am not sure that is could be a good 'getting started' documentation*". The goal of the generated documentation is not to replace the official one, which usually contains how-to-start documents, tutorials, or global explanations about the goal of the language. A subject also explained that he/she "*could not find any information about version ranges*" using TPD. The problem is that version ranges are not part of the language but a string convention defined by Eclipse. We think this is an issue of the language that makes use of a string to represent a concept. Moreover, two subjects suggested to provide hyperlinks to elements that an object can use (instead of referring to the elements parent to the current object). They think that this would help in understanding what a user can do with a given object.

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 [variable] names lack semantics [(*i.e., *meaning)], 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. Another subject pointed out that "*the generated documentation provides good examples but they lack at explaining what we can do with them*". We think that this point is important and depends on the models selected and used for the generation.

Fourth, the coding instructions provided in the generated documentation may not be useful. One subject explained that the "*typing instructions may be difficult to follow*". Another one said "*there is a lot of '*Type *}' in the text*". 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 *ThingML*. They identified different improvements to make in future work.

RQ4. We conducted a face-to-face discussion with one ThingML language designer (nicknamed as *LD1* hereafter). A second ThingML language designer (nicknamed as *LD2* below) gave us feedback by email. We got feedback from one TPD language designer by email (nicknamed as *LD3* below). *LD1* and *LD2* appreciated the generated documentation and consider it a useful complement of the official documentation, which is tedious to maintain on different supports (property #3: multi-platform documentation). LD1 noticed that each class parameter has, by default, its dedicated documentation. LD1 considers this choice as not always relevant, preferring to have a single piece of documentation that includes the optional attributes. LD2 explained that some attributes have a strong impact on ThingML and should have their own documentation, contrary to some other attributes. LD2 also asks for customization mechanisms to order or change the documentation generation. We explain to 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 piece of documentation. LD2 does not consider the typing instructions useful. This is in line with the comments of the subjects we discussed in RQ3. LD2 also highlights various issues in the documentation related to the action language of ThingML. We think this is a current limitation of the proposal, as discussed in Section 4.3: the ThingML action language complexifies its grammar to overcome left-recursion issues. Our prototype may fail in such situations, where a gap exists between the metamodel and the grammar. *LD3* found the contextual documentation improves the usability of the editor and suggested to contribute to Xtext on this point. As a language designer of TPD, *LD3* did not find the documentation useful for his needs since he already knows the concepts of the DSL. We agree since the approach aims at helping non-expert DSL end users. *LD3* also explained that the approach may be useful for DSLs having a large metamodels. *LD3*, however, considered that designing a large metamodel for a DSL (*e.g.,* UML, around 1.000 model elements [31]) in a bad practice since a DSL

should remain small. RQ1 shows benefits regarding the correctness of the created ThingML models when using the approach while ThingML contains 55 model elements, much less than UML.

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

4.2.4. Threats to validity

Internal validity. The obtained results may depend on the quality of the metamodel documentation provided by the language designers. To mitigate 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 regards 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 subject population validity, we asked the subjects to estimate 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. However, we 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.

4.3. Limits of the approach

The proposal has the following limitations. As discussed in Section 4.1, the proposed approach does not support constraints that language designers may define to precise the semantics of the metamodel. 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, as discussed in Section 4.1 too, on metamodel changes the activity diagram has to be re-generated. If a language designers changed this activity diagram, the changes will be lost. Approaches that treat the co-evolution of models have been proposed and could be used to overcome this current limitation of our proposal [32, 33].

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 remove this limitation.

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. Slicing techniques have been proposed to ease the understanding of large UML models [31]. Such techniques could be used to apply the approach on small separate parts of the metamodel rather than on the whole metamodel in one shot.

5. Related work

Software documentation. Because software languages are software too [17], 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 [34]. These approaches require a large data set to analyze. DSLs are dedicated to a specific and thus limited audience. The quantity of artifacts created from these DSLs may not be large enough to be mined to infer documentation. Manually-authored API documentation may be incomplete or wrong, and it should be complemented with the output of automated techniques [34]. 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 [35]. They argue that UML tools should be improved to support the co-evolution of the code and UML models. Our approach supports that principle by allowing developers to re-generate the documentation with little maintenance cost: only a manually-modified activity diagram may have to be updated.

DSL documentation generation. Meta-metamodel languages provide several tools to generate documentation. *EcoreDoc*[12] and *Ecore Documentation Generator*[13] are tools similar to JavaDoc but for the Ecore meta-metamodel language. These two tools generates documentation in different formats (HTML, LaTeX) based on the information embedded in an Ecore model (*i.e.,* a metamodel).

MetaEdit+ also provides a documentation generation tool.[14] This documentation tool generates from a MetaEdit+ project, pieces of documentation (one for each model that the project contains). Each piece of documentation contains a picture of the model (if the model has a graphical syntax), meta-information related to the model, and navigation links to the other pieces of documentation of the project.

These tools provide a basic support for helping language designers in documenting DSLs. *Docywood* processes input models to provide illustrative examples. *Docywood* relies on model slicing techniques to decompose documentation into pieces of documentation that each focuses on a specific concept of the DSL.

Automated exercise 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 [36] or to illustrate mathematical problems [37]. In MDE, Gómez *et al.* propose an approach that generates exercises to train users in recognizing well-formed and badly-formed models [38]. To this end, their approach generates models that focus on a targeted concept (*e.g.,* state machines) and mutates them to get badly-formed models. Our method could improve exercises generation by introducing the notion of exercise 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 from such concepts, for example to train on metamodel elements following a progressive level of difficulty. It would be interesting for language designers to provide some information to guide the exercises generation order, like the UML activity diagram of our approach.

Automated models production. In MDE, model generation and mutation 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 [39, 40]. Metamodel coverage can be performed at two levels [40]: class coverage and cardinality coverage (*i.e.,* attributes and associations cardinalities). Regarding cardinality coverage, the techniques proposed consist in designing partitions over cardinalities (*e.g.,* $[0..*]$ is partitioned to $\{\{0\},\{1\},[2..*]\}$) [41, 42]. 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 [43]; translate this objective to a multi-function optimization problem [44]. As explained in Section 3.4.1, our approach is based the following coverage criteria: each class, attribute, and reference is covered by one piece of documentation. Several metamodel elements can be covered in the same piece of documentation.

[12] `https://marketplace.eclipse.org/content/ecoredoc`

[13] `https://marketplace.eclipse.org/content/ecore-documentation-generator`

[14] `https://www.metacase.com/support/55/manuals/meplus/Mp.html#Mp-5_2_2.html`

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 [44]. 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 documentation 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 documentation.

Decision making. The automatic production of documentations is related to the design of auto-completion and related techniques that help users while creating models. Proactive modeling aims at helping users while creating models [45]. This technique complements ours since it proposes to users actions to perform while creating graphical models. As with 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 [46]. Domain analysis can be used to get, in the context of software language engineering, domain models (*e.g.,* metamodels) [18]. Approaches have been proposed to transform use cases into domain models [47]. 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.

Empirical studies on DSLs. Johanson *et al.* detailed an empirical study that focuses on the potential benefits the use of a DSL may have compared to the use of a GPL (*General Purpose Language*) [48]. More precisely, the study focuses on a domain expert audience not familiar with programming. These domain experts used a DSL for high-performance marine ecosystem and C++ to perform representative tasks of their domain. The effectiveness and efficiency of the DSL and C++ were then measured and compared. The study exhibits significant results in favor of the DSL: higher accuracy, less time spent by domain experts.

Häser *et al.* conducted a study to compare the use of a DSL for BDD (*Behaviour Driven Development*) to another version of the same DSL, extended with business model concepts [49]. The results show that test case creation is significantly faster with the DSL supporting business model features, and that users feel better supported by this extended DSL.

Albuquerque *et al.* proposed an approach to compare the usability of DSLs in the context of DSL maintenance [27]. Their approach is based on the CDN (*Cognitive Dimensions of Notations*) framework [50], which defines several cognitive dimensions (*e.g.,* consistency and error-proneness) to evaluate the usability of notational systems, such as DSLs. They identified two main cognitive dimensions to take into account: expressiveness and conciseness, that are subdivided into other cognitive dimensions from the CDN (eight dimensions in total, *e.g.,* capacity of abstraction for expressiveness and viscosity, or adaptability, for conciseness). The results of their evaluation show that these dimensions are critical and useful to evaluate a DSL usability at an early stage of its development.

Barišić *et al.* proposed a formal model to evaluate DSLs [51]. This model is designed to comply with the approach of iterative user-centered evaluation practices. The philosophy of this approach is to evaluate the DSL at several stages of its development in order to avoid performing only a last-minute evaluation, when it is too late to implement subjects' reviews. Our evaluation complies with the model. It is compatible with iterative evaluation principles since subjects reviews can be taken into account in future works.

More recently, Barišić *et al.* proposed the USE-ME framework to design usability studies of DSLs and their related tools [28]. This framework allows to formally model the context, goals, and evaluation protocol of a DSL or tool under study. They evaluated their approach by selecting four DSLs and making master students use the framework to model an appropriate usability study. For each DSL, they measured the correctness of the produced models in time-limited sessions and asked feedback from the participants. The results show significant satisfactory results for a small limited time (12h in total for the whole process, from context definition to protocol modeling). In our approach, we also use the time spent and the correctness as evaluation metrics.

De Sousa *et al.* presented an approach to design spatial simulation scenarios with the DSL3S language [24]. A usability evaluation was conducted. This evaluation is partially based on a questionnaire with Likert-like questions. We also submitted a questionnaire to the DSL designers. The difference is that our questions are open and not in a Likert-scale format.

Giraldo *et al.* presented two literature reviews in the field of assessing the quality of MDE methods, on academic and industrial papers, webpages, blogposts, working notes, *etc.* [23] The authors show that quality concerns expressed

by the MDE practitioners are not taken into consideration when designing frameworks for MDE quality evaluation. In particular, the Physics of Notations (PoN) framework lacks of crucial metrics. The main supported idea in this paper is that academic research should show more interest in industrial constraints and organizations in order to propose MDE methods and tools that could really be used. In our work, this principle is applied by asking questions about the usage of *Docywood* to the DSL designers (to answer RQ4).

6. Conclusion and Future Work

6.1. 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, both dedicated to a computer science audience. The evaluation shows that the proposed approach improves the correctness of *ThingML* models created by novice end users. *ThingML* is a textual, large metamodel-based (96 meta-classes) DSL. The evaluation does not expose benefits for the *Eclipse Target Platform* DSL, which is a small (eight meta-classes) textual and metamodel-based DSL. We thus think that *Docywood* may be useful to document large textual DSLs dedicated to a computer science audience. Feedback from end users and language designers exhibits qualitative benefits of the proposal with regard to the DSL documentation challenges. They identify several shortcomings such as: coding instructions that may not be useful; hyperlinks to the elements that the current object can use should be provided; the readability of the generated documentations strongly depends on the example selected by the language designers and the readability of the metamodel documentation; the documentation may have a limited interest for small DSLs and expert end-users. 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.

6.2. Future Work

The current proposal has several limits that we will study in the future. First, we will investigate how to encompass DSLs with concrete graphical syntaxes. We will also address 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 to start using this DSL and our generated documentation. Finally, we will investigate to what extent a language designer can define her own activity diagrams to design specific documentation.

Acknowledgments

We thank the ThingML language designers 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 37 (2005) 316–344. doi:10.1145/1118890.1118892.

[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). doi:10.1109/MS.2009.92.

[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: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, SLE 2015, 2015, pp. 25–36.
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, 2013, pp. 272–287.
URL https://hal.inria.fr/hal-00818943

[8] T. Kosar, S. Bohra, M. Mernik, Domain-specific languages: A systematic mapping study, Information and Software Technology 71 (2016) 77–91. doi:10.1016/j.infsof.2015.11.001.
[9] M. Fowler, Language workbenches: The killer-app for domain specific languages (2005).
[10] M. Bravenboer, E. Visser, Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, 2004, pp. 365–383. doi:10.1145/1028976.1029007.
[11] G. Uddin, M. P. Robillard, How API documentation fails, IEEE Software 32 (4) (2015) 68–75. doi:10.1109/MS.2014.80.
[12] J. Nielsen, Enhancing the explanatory power of usability heuristics, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '94, ACM, 1994, pp. 152–158. doi:10.1145/191666.191729.
[13] 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
[14] 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
[15] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.
[16] 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. doi:10.1145/2976767.2976812.
[17] 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. doi:10.1007/978-3-642-19440-5_21.
[18] A. Van Deursen, P. Klint, Domain-specific language design requires feature descriptions, CIT. Journal of computing and information technology 10 (1) (2002) 1–17. doi:10.2498/cit.2002.01.01.
[19] J.-P. Tolvanen, S. Kelly, Metaedit+: Defining and using integrated domain-specific modeling languages, in: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, 2009, pp. 819–820. doi:10.1145/1639950.1640031.
[20] M. Weiser, Program slicing, in: Proceedings of the 5th International Conference on Software Engineering (ICSE'81), IEEE Press, 1981, pp. 439–449.
URL http://dl.acm.org/citation.cfm?id=800078.802557
[21] F. Shull, J. Singer, D. I. Sjøberg, Guide to advanced empirical software engineering, Springer, 2007.
[22] V. R. Basili, G. Caldiera, H. D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering 2 (1994) 528 – 532.
[23] F. D. Giraldo, S. España, Ó. Pastor, W. J. Giraldo, Considerations about quality in model-driven engineering, Software Quality Journaldoi:10.1007/s11219-016-9350-6.
URL https://doi.org/10.1007/s11219-016-9350-6
[24] L. M. de Sousa, A. R. da Silva, Usability evaluation of the domain specific language for spatial simulation scenarios, Cogent Engineering 5 (1). doi:10.1080/23311916.2018.1436889.
URL http://doi.org/10.1080/23311916.2018.1436889
[25] M. Challenger, G. Kardas, B. Tekinerdogan, A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems, Software Quality Journal 24 (3) (2016) 755–795. doi:10.1007/s11219-015-9291-5.
URL https://doi.org/10.1007/s11219-015-9291-5
[26] G. Kahraman, S. Bilgen, A framework for qualitative assessment of domain-specific languages, Software & Systems Modeling 14 (4) (2015) 1505–1526. doi:10.1007/s10270-013-0387-8.
URL https://doi.org/10.1007/s10270-013-0387-8
[27] D. Albuquerque, B. Cafeo, A. Garcia, S. Barbosa, S. Abrahão, A. Ribeiro, Quantifying usability of domain-specific languages: An empirical study on software maintenance, Journal of Systems and Software 101 (2015) 245 – 259. doi:10.1016/j.jss.2014.11.051.
[28] A. Barišić, V. Amaral, M. Goulão, Usability driven DSL development with USE-ME, Computer Languages, Systems & Structures 51 (2018) 118 – 157. doi:https://doi.org/10.1016/j.cl.2017.06.005.
URL http://www.sciencedirect.com/science/article/pii/S1477842417300477
[29] I. S. MacKenzie, Human-computer interaction: An empirical research perspective, Newnes, 2012.
[30] D. J. Sheskin, Handbook Of Parametric And Nonparametric Statistical Procedures, Fourth Edition, Chapman & Hall/CRC, 2007.
[31] 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
[32] 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. doi:10.1007/978-3-642-36089-3_9.
[33] 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. doi:10.1109/EDOC.2008.44.
[34] 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.
[35] 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. doi:10.1109/TSE.2006.59.
[36] 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. doi:10.1145/2530544.2530546.
[37] O. Polozov, E. O'Rourke, A. M. Smith, L. Zettlemoyer, S. Gulwani, Z. Popovic, Personalized mathematical word problem generation, in: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, 2015, pp. 381–388.

[38] 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 49 (2016) 152 – 173. doi:10.1016/j.cl.2016.11.001.

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

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

[41] 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/

[42] 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. doi:10.1007/s10270-007-0074-8.

[43] 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. doi:10.1109/MODELS.2015.7338262.

[44] 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. doi:10.1145/2976767.2976785.

[45] 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.

[46] I. Jacobson, Object-oriented development in an industrial environment, in: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications, ACM, 1987, pp. 183–191. doi:10.1145/38765.38824.

[47] 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. doi:10.1007/978-3-642-04425-0_37.

[48] A. N. Johanson, W. Hasselbring, Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment, Empirical Software Engineering 22 (4) (2017) 2206–2236. doi:10.1007/s10664-016-9483-z.

[49] F. Häser, M. Felderer, R. Breu, Is business domain language support beneficial for creating test case specifications: A controlled experiment, Information and Software Technology 79 (2016) 52 – 62. doi:10.1016/j.infsof.2016.07.001.

[50] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roes, A. Wong, R. M. Young, Cognitive dimensions of notations: Design tools for cognitive technology, Cognitive Technology (2001) 325 – 341doi:10.1007/3-540-44617-6_31.

[51] A. Barišic, V. Amaral, M. Goulão, B. Barroca, Evaluating the usability of domain-specific languages, in: Software Design and Development: Concepts, Methodologies, Tools, and Applications, IGI Global, 2014, pp. 2120–2141.