Textual Use Case DSL with Sequence Diagram Transformation

Andrés Carrasco

Andres.Carrasco@student.uantwerpen.be

University of Antwerp

Abstract

Even though visual languages have been often regarded as better suited for some Domain Specific Languages (DSL), we explore textual DSLs with the Xtext framework by creating a DSL for use cases and generating from them sequence diagrams through a model-to-model transformation and a subsequent model-to-text transformation with the tools provided by Xtext.

Keywords: textual languages, Xtext, domain specific languages, DSL, sequence diagrams, use case

1. Introduction

General Purpose Languages (GPL), e.g. Java, are designed for solving any kind of problem. However, due to their vast scope, in certain domains the solutions can become quite complex. For example, performing a query in a database in pure Java without using the domain specific language SQL, a big overhead would be added in parsing and interpreting the data coming out and to the database. Whereas simply using SQL reduces this overhead completely. SQL is just one example where a domain specific language (DSL) is more appropriate than a GPL. Nevertheless, a DSL does not always replace a GPL, as the DSL is often interpreted or compiled into a GPL. Therefore, the DSL is being utilized for simplifying the complexity that a GPL might impose on a problem.

Both SQL and Java are textual languages, nevertheless, not all DSL are textual. Depending on your problem scope, a visual DSL might be better suited than a textual DSL, in some cases even a mixture of both. However, it is often argued that visual languages provide a better overview and are preferred above textual languages, as they are often referred to as more intuitive.

Regardless of their advantages, they both have a set of requirements which makes the developing of DSLs expensive. *Xtext* [1] is a framework for developing

[1] http://www.eclipse.org/Xtext/

textual GPLs and DSLs, which aids by creating all of the infrastructure and thus fulfilling all of their requirements.

In this writing, we explore the possibility of creating a textual DSL for use cases with Xtext. A *use case* is a technique utilized for analyzing requirements in software engineering, composed of a list of actions that define the interaction between actors. Through this list they help discover, and most importantly document, the functional requirements of a system.

Alongside designing a DSL for use cases, we also explore the possibility to translate it to a sequence diagram. A *sequence diagram* is a diagram that shows the behavior of a system in terms of objects an their ordered interaction between each other. Utilizing both diagrams, the requirements of the system's design could be revised in different formalisms.

In Section 2 we give a short comparison of textual and visual languages. Next in Section 3 we discuss the creation of a DSL with Xtext. In Section 4 two possible approaches to controlling natural languages are presented. In both Section 5 and Section 6 the components of use cases and sequence diagrams are discussed respectively. Afterwards, in Section 7 we discuss our approach of constructing a DSL and its translation to a sequence diagram. In Section 8 we give a short example of a use case and its subsequent generated sequence diagram. Lastly in Section 9 we give our conclusions and future work.

2. Visual vs Textual Languages

In their article Grönninger et al. (2014) [1] argue that textual languages have the following advantages over visual languages:

1. **Content compactness.** Grönninger et al. (2014) argue that graphical languages require more space to convey the same amount of content, thus when selecting an appropriate language type for a DSL, in which space is critical, textual languages have a clear advantage.
2. **Speed of creation.** the authors also argue that current text editor technologies are more efficient than visual editor technologies, due to their nature they often constrain the designer, leading to time loss.
3. **Integration of Languages.** in this point, Grönninger et al. (2014) argue that integrating different types of languages is easier if they are textual, as doing so in visual languages leads to a lack of developer efficiency.
4. **Speed and quality of formatting.** formatting text is a trivial task for standard formatting algorithms. In contrast to this, formatting a visual model is not, as depending on the semantics there can be a special layout thought of, probably requiring a custom layout algorithm.
5. **Platform and tool independency.** textual languages have the advantage that they can be manipulated by any text editor available, this gives them a high degree of flexibility, whereas visual languages usually need their own environment.
6. **Version control.** the last advantage Grönninger et al. (2014) point out, is the possibility of using standard version control systems. Version control

systems are widely used for text, whereas with visual languages they have not, as they have not proven to be very reliable.

Although some of the previous points seem to be very convincing, in some cases visual languages might still be a better choice. Whenever deciding between one or the another, all advantages and disadvantages should be considered, as to decide on the best type for the proposed DSL.

3. Creating a DSL with Xtext

A DSL requires the ability to read the input text, parse it, process it and possibly interpret or generate code in a GPL. However, as most programming languages a DSL also needs to have a good IDE support, syntax highlighting, continuous background parsing, error markings in the input text, auto-complete features, hyperlinking between references, and even possibly suggesting quick-fixes, among other functionalities [2].

As it seems so far, developing a DSL is also quite a task, here is where Xtext [2] comes in handy. Xtext can generate all of the requirements of DSLs automatically, after specifying grammar rules utilizing their grammar language. Moreover, Xtext is able to create a compiler from your DSL to any other language. Xtext also provides special support for targeting the Java Virtual Machine through Xbase. In other words, utilizing the Xtext framework, reduces the overhead of creating a DSL and potentially enabling your DSL to unlock its full potential. Xtext makes it specially easy, because of their well chosen default configuration, which usually covers all of the common needs, however it is completely configurable.

Xtend itself uses a grammar language based on the Eclipse Modeling Framework (EMF). The input grammar is parsed using Another Tool For Language Recognition (ANTLR). The latter is widely used by programming language like Java and Python. From the grammar parsed by ANTLR a new linker and parser are created for the language specified by itself, alongside other popular IDE features. Creating thus a full blown DSL with IDE support.

4. Controlled Natural Language

Use cases are written using informal communication: with a natural language. The use cases resulting from this process usually must be used on human review processes, resulting to high costs or reduced quality, due to their static nature [3]. Knauf [3] proposes to control the quality of the natural language by using templates, also known as *boilerplates*, therefore removing the need for human interaction. Li [4], proposes to manually normalize the natural language for then to be input into a parser. However, this approach still requires manual labor and might still retain some of the previous mentioned disadvantages to some degree.

[2]http://www.eclipse.org/Xtext/

5. Use Case

A use case is typically composed of the following components:

- **Name**: usually a phrase that grasps the goal of the primary actor.
- **Scope**: defines the scope of the use case.
- **Granularity Level**: defines the level of detail of the use case. There are usually three granularity levels. The *summary level* defines actions that encompass multiple lower-level use cases. While the *user goal level* specifies the actions of an actor. Lastly, the *subfunction level* specifies actions that support the user goal level use case.
- **Intention**: specifies the intention of the primary actor in their context.
- **Multiplicity**: specifies the number of different components.
- **Actors**: are the entities that interact within or with the system.
- **Main Success Scenario**: define the sequence of interaction steps that conforms to a successful scenario.
- **Extensions & Exceptions**: define additional or alternative interaction steps for the mains success scenario.

This, however, is only a typical format for use cases, there are different formats that include different components.

6. Sequence Diagram

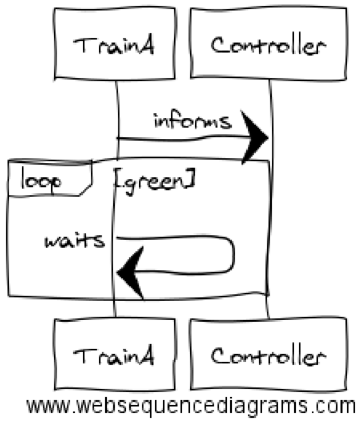


Figure 1: An Example Sequence Diagram

A Sequence Diagram is composed by the representation of entities interacting with each other. An example can be seen in Figure 1. They are composed of *boxes* containing the name of the entity they represent with a vertical line representing their *lifeline*. To show communication between the different entities, an arrow is drawn from one entity to another. These arrows are called messages and there are two types: a *message*, represented by an arrow, and a *reply*, represented by an arrow with a striped line. Both types of arrows can be seen in Figure 2, the message up top and reply below. Moreover, when an arrow has an open arrow head, it is meant to be asynchronous. All other arrows are meant to be synchronous.

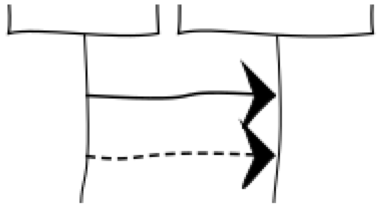


Figure 2: Different types of synchronous messages

Alongside the messages, there are different types of constructs called *combined fragments*. These are drawn as a square above the lifelines of the entities, with a square on the upper left corner with the type as shown in Figure 3. There are many different types of combined fragments, however, we will only focus on the following:

- **Alt**: this operator represents two or more alternatives of interactions, on each of the alternatives a guard condition is set to specify when this alternative fragment is to be active.
- **Opt**: this operator represent optional interactions, this also has a guard condition which specifies when is this fragment active.
- **Loop**: this operator represents iterative interactions which repeat until the guard condition is not met anymore.

Other constructs such as break, seq, par, strict, etc. are not being considered in this writing.

6.1. Transformation to WebSequenceDiagrams.com Syntax

Our goal is to eventually transform to a sequence diagram, for this we need an engine that can generate visual representations of a sequence diagram given a specification. For this we selected the WebSequenceDiagrams.com[3] tool. Unfortunately, this tool does not support all of the constructs of a sequence diagram, therefore we only created transformation for the supported components.

[3]https://www.websequencediagrams.com/

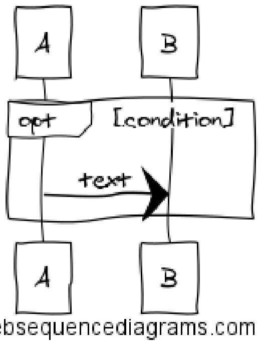


Figure 3: The *Opt* combined fragment

7. Creating the Use Case DSL with Xtext

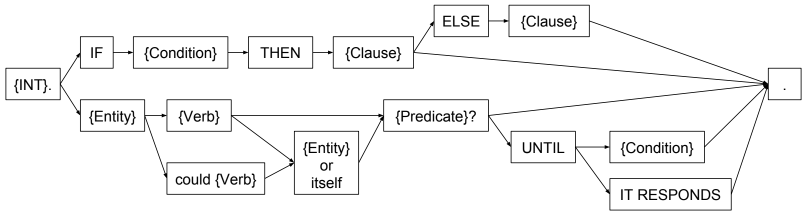


Figure 4: The proposed grammar for an interaction step.

In order to create the DSL with Xtext, a grammar must be designed that will parse the use case format specified in Section 5. The grammar will be conformed of all of the elements specified in the format. However, a distinction between the primary actors and secondary actors will be made with the keyword `actor` and `secondary` respectively.

The workflow depicted in Figure 5 will be implemented, consisting of a parsing step, a model-to-model (M2M) transformation step, and finally a model-to-text (M2T) code generation step.

The first step in implementing this workflow would be to specify a grammar, using Xext's grammar language. The most interesting piece of this grammar, would be the steps in the scenario, as here will the boilerplates be used to control the natural language. A representation of the proposed grammar can be seen in Figure 4. In the figure, the following constructs can be seen: `{INT}`, `{Condition}`, `{Verb}`, `{Entity}`, `{Predicate}`, and `{Clause}`. Each of them represent a placeholder for different types of data. `{INT}` represents a placeholder for an integer, whereas `{Predicate}`, `{Verb}` and `{Condition}` represent a placeholder for a string. In contrast to this, the placeholder `Entity` expects either an actor or a secondary actor, which in turn has to be specified before in

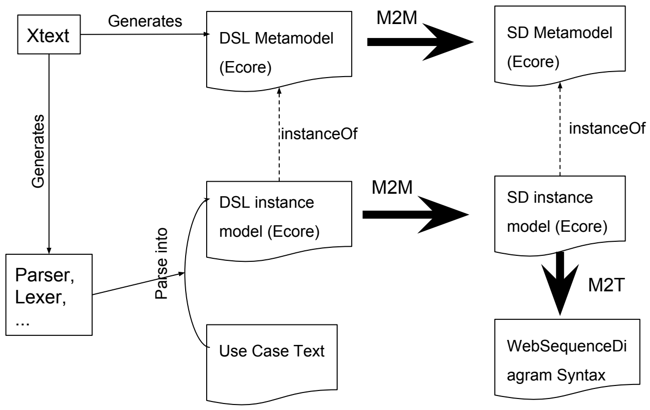


Figure 5: The proposed workflow.

the grammar. Although, after the verb the keyword `itself` can be alternatively used to refer to the same first entity. Lastly, the placeholder `Clause` expects any input of a interaction step without the numbering and punctuation at the end. This allows to have nested constructs within the alt operator for example.

Depending on the sentence construction, the interaction step will be translated into one of the previously described messages or combined fragments. For example, the sentence:

```
3. the System "informs" Teller "that deposit was
   successful and waits" until it responds.
```

Due to the `until it responds` keywords, it will be translated to a `BlockingSynchronous` construct. Subsequently it will be translated into the WebSequenceDiagram syntax, resulting in the Figure 6. We will discuss what the `BlockingSynchronous` is on the Subsection 7.2.

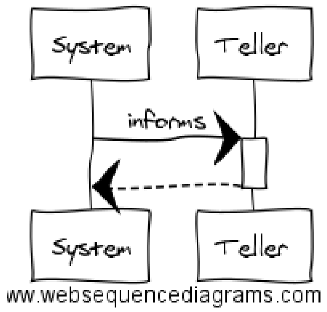


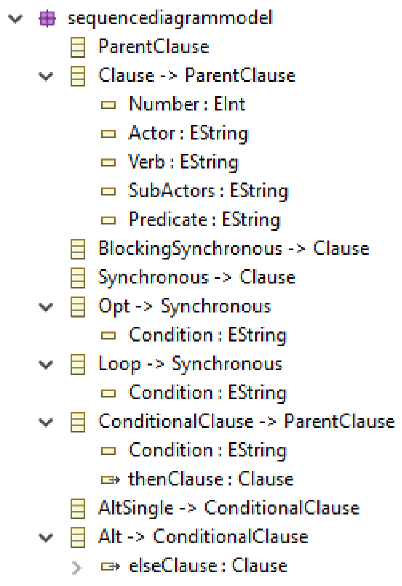
Figure 6: A BlockingSynchronous transformation example

7.1. *Parsing the input Text*

After parsing the use case with the supplied grammar, it is parsed into an *Ecore instance model*, specified by a *Ecore metamodel*. Both of them where generated automatically by Xtext when specifying the grammar.

7.2. *Model-to-Model Transformation*

Once the Ecore instance model is made available, we perform a model-to-model transformation. For this, we specified a target model with Ecore, representing the components of a sequence diagram. The target model can be seen in Figure 7.



```
sequencediagrammodel
    ParentClause
    Clause -> ParentClause
        Number : EInt
        Actor : EString
        Verb : EString
        SubActors : EString
        Predicate : EString
    BlockingSynchronous -> Clause
    Synchronous -> Clause
    Opt -> Synchronous
        Condition : EString
    Loop -> Synchronous
        Condition : EString
    ConditionalClause -> ParentClause
        Condition : EString
        thenClause : Clause
    AltSingle -> ConditionalClause
    Alt -> ConditionalClause
        elseClause : Clause
```

Figure 7: The target model grammar in Ecore.

Each of the components of the target Ecore model represent a specific construct of the sequence diagram. For instance, the `BlockingSynchronous` represents the construct seen in Figure 9. Similarly, `Synchronous` represents any simple message being sent from entity to entity, while `Loop`, `Opt`, `Alt` and `AltSingle` all represent the construct their name suggests. The `AltSingle` is meant to be used on an alt without an alternative, i.e. it only has one guard condition and interaction steps.

In order to perform the model-to-model transformation, the parser for the use case grammar had to be programmatically requested and subsequently transformed. To achieve this in Xtext, two classes were written: `UseCaseM2M.java` and `UseCaseToSequenceDiagram.xtend`. The Java class was utilized to traverse the parsed use case model, and implement the logic of the transformation to the Sequence Diagram model. To achieve this, method calls from the Xtend class were utilized, which depending on the method called instantiates Ecore objects with the given parameters, essentially performing the model-to-model transformation.

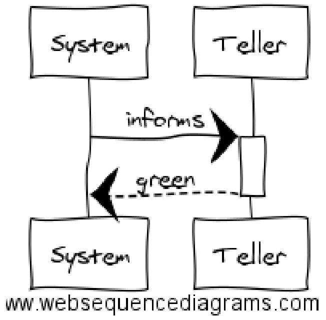


Figure 8: The BlockingSynchronous visual representation.

7.3. Code Generation

As a last step, the given sequence diagram Ecore model will be generated into the syntax of WebSequenceDiagram.com. To achieve this, through the `UseCaseDSLGenerator.xtend` a code generation model-to-text method is implemented, which essentially creates the syntax depending on the Ecore instances. For the user's ease, a context menu entry was added when right clicking the use case editor, which upon selecting, all of the workflow will be performed and the resulting code persisted in the file `/src-gen/generated.seq`.

8. Example

In this section, we show an example of a written use case and its translation to a sequence diagram:

```
useCase: "Deposit Money"

scope: "Bank Accounts and Transactions System"

actor: Teller

secondary: System

intention: "The intention of the Client is to deposit
    money on an account.
Clients do not interact with the system directly; instead
    , for this use case,
a Client interacts via a Teller"

```

```
level: "User-Goal level"

multiplicity:
"Many Clients may be performing deposits at any one time.
A Client only requests one deposit at a given time."

scenario: {
        1. Teller "requests" the System "to deposit money
             on an account, providing sum of money".
        2. the System "validates" itself "the deposit,
            credit account with the requested amount,
            records details of the transaction".
        3. the System "informs" Teller "that deposit was
            successful and waits" until it responds.
}
```

The resulting sequence diagram is the following:

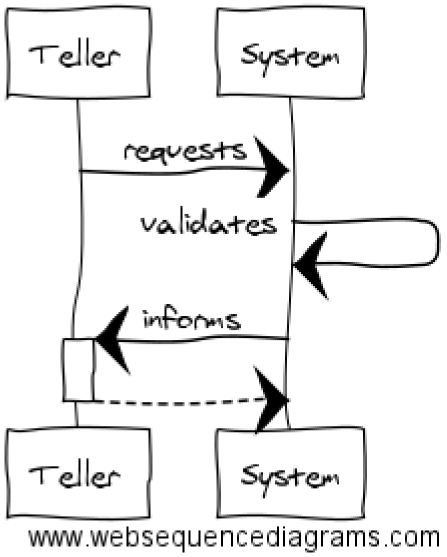


Figure 9: The reslting sequence diagram.

9. Conclusions and Future Work

After creating the DSL and the subsequent necessary steps to create a transformation, we were satisfied with the ease Xtext provides for making changes without breaking functionality. Moreover, knowing that Xtext takes care of all the infrastructure we could focus entirely on the specification of a functional DSL and proper translation into a sequence diagram.

However, due to the reduced number of features WebSequenceDiagrams.com provides, we were unable to present a complete DSL for all of the possible sequence diagram's constructs. Notwithstanding, our DSL also came short with some of the possibilities both sequence diagrams and WebSequenceDiagrams provided. Such as taking notes, and having multiple alternatives in the Alt construct. Moreover, due to a problem with the generation of the lexer and parser, we were unable to include a preposition (the, to, a, etc.) to the `actor`, as it didn't generate any of them when both the `actor` and `secondary` had them. In the future support for this could be added, as well as utilizing a different sequence diagram tool that supports all of the components sequence diagrams provide. Lastly, support for more than one nested clause in the `if then else` construct should also be provided.

References

[1] H. Grönninger, H. Krahn, B. Rumpe, M. Schindler, S. Völkel, Textbased Modeling, ArXiv e-prints`arXiv:1409.6623`.

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

[3] C. Knauf, Xtext and controlled natural languages for software requirements (jul 2016).
URL `https://blogs.itemis.com/en/xtext-and-controlled-natural-languages-for-software-requirements-part-1`

[4] L. Li, Translating use cases to sequence diagrams, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 293–.