Towards the Automated Conversion of Natural-Language Use Cases to Graphical Use Case Maps

By
Jason Kealey
2657431
jkealey@shade.ca

Report presented to:
Dr. Inkpen
for the course
CSI5180: Topics in Statistical Natural Language Processing

SITE, University of Ottawa
December 5th, 2005

Table of contents

Introduction 3
Background Information 3
Use Cases 3
Use Case Maps 4
jUCMNav 5
UCEd 5
Problem Statement 5
Review of related literature 6
Methodology 8
Data 9
Parameters 10
Evaluation Methodology 10
Result Description 10
1) Actors 11
2) Primary successful scenario 13
3) Alternatives 14
4) Redirection actions 16
5) Use case inclusion 17
6) Use case extensions 19
7) Various conditions 20
8) Time constraints 21
9) Any * alternatives 23
10) Operation effects 24
Results Discussion 24
Relevance 24
Future Work 25
Conclusion 25
References 26

Introduction

This report presents my CSI5180 project: the automated conversion of textual use cases into Use Case Maps. When implementing my project, many issues came up and require further analysis and discussion with domain experts such as Daniel Amyot. Given this is a course project, we did not stop to reflect on the various issues because these reflections would have considerably slowed down the project. The prototype tool was completed with little or no supervision or validation. The purpose of this report is to present the tool to domain experts to initiate discussions about the encountered issues. However, we also try to put emphasis on the NLP aspects of the project and attempt to include as much information as possible to bring Use Case Map novices up to speed. For more information, see [1].

Background Information

Use Cases

One of the most fundamental steps in today's software engineering processes is the elicitation of use cases. Because they are written in natural language, use cases are easy to understand by all stakeholders. Use cases illustrate a scenario of interactions between an actor (or multiple actors) and a system from a black-box perspective. By their very nature, use cases are informal and are most useful to obtain high-level views of the function behaviour of a system.

Figure 1, taken from *Behavior Specifications from Textual Use Cases [6]*, illustrates a typical use case. We see a black-box view of a Seller interacting with a Marketplace Information System, including different extensions and alternatives.

The UML notation provides a graphical view called a Use Case Diagram, which represents how different Use Cases interact with each other, as inclusion and inheritance are supported. An example of such a Use Case Diagram taken from the UCEd user manual [10] is given in Figure 2.

Use Case: M1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Primary Actor: Seller
Supporting Actor: Trade Commission
Main success scenario specification:
1. Seller submits item description.
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history
6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with a uniquely identified authorization number.
Extensions:
2a Item not valid
2a1 Use case aborts
5a Seller's history inappropriate
5a1 Use case aborts
6a Trade commission rejects the offer
6a1 Use case aborts
Variations:
2b Price assessment available
2b1 System provides the seller with a price assessment.

Figure 1: Textual Use Case "Seller submits an offer"

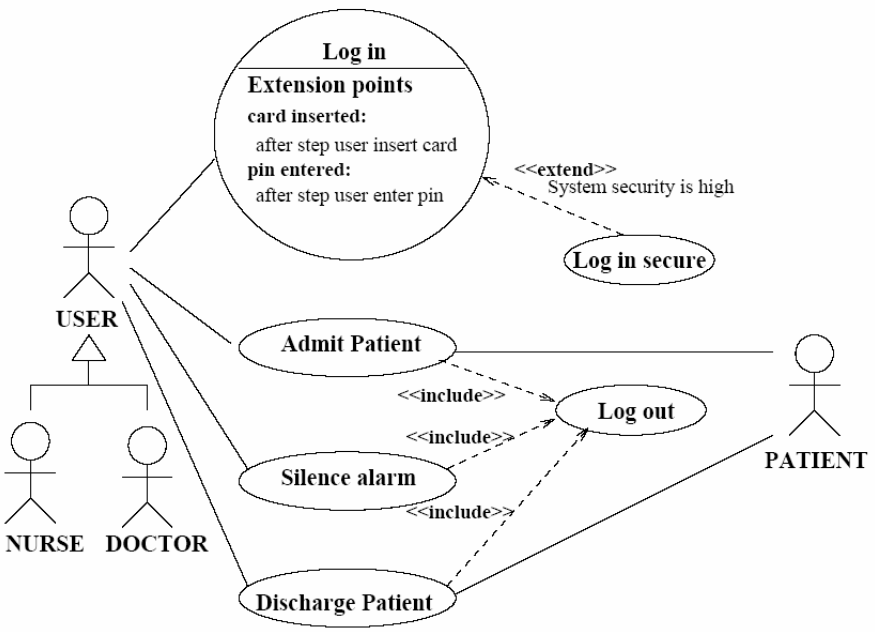


Figure 2: Sample Use Case Diagram

Use Case Maps

The Use Case Map (UCM) notation is a part of the User Requirements Notation (URN) [1] currently undergoing standardization by the ITU. UCMs illustrate visually scenarios cutting through a system's component structure. Because of their visual cleanliness and apparent simplicity, UCMs are quickly picked up by all stakeholders. Contrasting with textual use cases, UCMs present a gray-box view of a system. Furthermore, being a structured notation with a specified syntax, one can perform various different tasks on UCM instances. For example, UCMs have already been used for test case generation, scenario generation, performance modelling, and the visualisation of existing software.

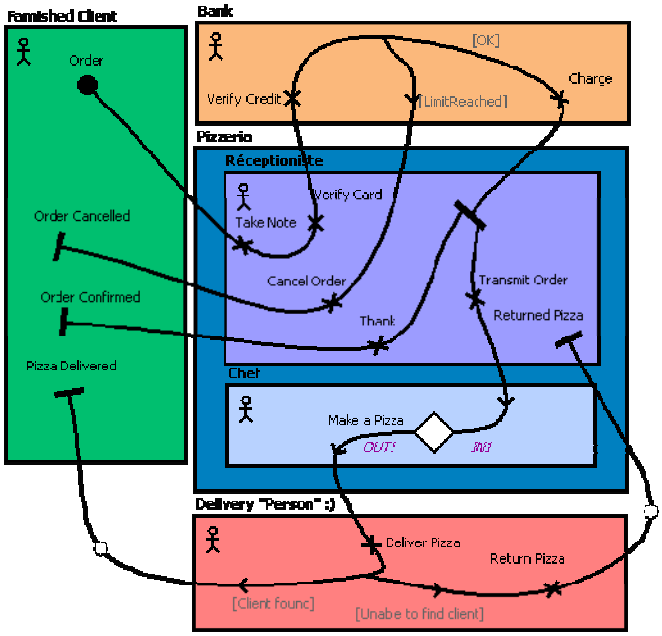


Figure 3: Sample Use Case Map

jUCMNav

jUCMNav is an Eclipse plugin (Java based) visual editor for the Use Case Map notation [5]. It is destined to replace UCMNav, the original UCM editor. jUCMNav was the subject of my software engineering capstone project, under the supervision of Daniel Amyot. Our tool makes use of the Eclipse platform and its plugins, such as the Graphical Editing Framework (GEF) and the Eclipse Modelling Framework (EMF). For more information, see http://www.softwareengineering.ca/jucmnav.

UCEd

UCEd [10] uses an adhoc natural language processing implementation to parse the Use Cases written using the tool and transform them into a more formal representation. UCEd allows software engineers to add more information than what would conventionally be expected, mainly through its domain model. Because of this, it can generate state models that realize the use cases and perform simulations on these state models [11]. Although the subset of English that is allowed in the tool is very limited, it represents a good portion of well-written use cases. UCEd is also implemented in Java and relies on the Eclipse framework.

Problem Statement

Both use cases and Use Case Maps are used in the early stages of software development. Use Cases are widely used and closely related to Use Case Maps. However, no tool has been developed to facilitate the generation of UCM from Use Cases.

UCMs are a very interesting notation because they are more flexible than UML activity diagrams and because they remain very easy to manipulate and understand. UCMs allow for a quick evaluation of different architectures and scenarios. Combined with GRL, the Goal-Oriented Requirement Language that is also a part of URN, UCMs help bridge the gap between textual requirements and system design [2]. Different behavioural and architectural alternatives can quickly be modeled visually.

The project presented in this report does not try to validate the argument that UCMs are useful requirements engineering artefacts. We tackle a smaller problem. The notation, still under development, has not yet benefited from widespread adoption. One of the main goals of jUCMNav was to enhance the usability of the original UCMNav tool. We believe that the lack of usable tools had a dissuasive effect on the adoption of the notation. With this project, we wish to continue to knock down any barriers that hinder the propagation of the notation. The problem we wish to address here is to provide mechanisms to help integrate UCMs in the development process of more software engineers. How will we achieve this?

The implementation of a bootstrapping mechanism that generates UCMs from textual use cases, which are already being widely used, will help leverage the work already produced by software engineers. The main idea is to provide the means to try out the notation on non-trivial examples and play with it without having to build everything from scratch.

Obviously, for a course project, it would be impossible to implement anything that would work on textual use cases directly and import those into jUCMNav.

Therefore, our high-level goals are as follows:

- Study the NLP related to understanding the different use cases constructs.
- Express improvements to be made to the NLP tool used in UCEd.
- Define mappings between use case concepts and UCM concepts.
- Prototype the conversion utility from UCEd use cases to jUCMNav UCM.

Our goal for this course project is not to seamlessly integrate both tools but rather to experiment with the conversion process and the NLP involved. Later, possibly during my master's thesis, this prototype will be integrated into jUCMNav. Furthermore, I am also interested in implementing the inverse process, generating textual use cases from use case maps. Since the former are more widespread, some companies might want to revert to the textual notation when dealing with other professionals.

Review of related literature

When talking about use cases, *the* reference on the subject is unequivocally *Writing Effective Use Cases* [3] by Alistair Cockburn. The book offers interesting insight on the nebulous subject of use cases. Anyone who has ever written use cases quickly finds that they aren't sure they are at the correct abstraction level or if they are writing use cases in the correct format. Albeit no format is actually imposed for writing use cases, people tend to adhere to a certain style. Along with different use case writing templates, Alistair Cockburn presents guidelines that should be followed to maximize the benefit of use cases and answers many questions that come to mind when first introduced to use cases. This project is not about writing use cases per se, but since the UCEd use case template is inspired by this book, it is still a great reference.

One passage that is relevant to our work is as follow:

> *A use case details the interactions and internal actions of actors, interacting to achieve a goal. A number of diagram notations can express these things: sequence charts, collaboration diagrams, activity diagrams, and Petri nets. If*

you choose to use one of these notations, you can still use most of the ideas in this book to inform your writing and drawing.

The graphical notations suffer from two usability problems. The first is that end users and business executives are not likely to be familiar with the notations, and have little patience to learn. Using graphical notations means you are cutting off valuable readers.

The second problem is that the diagrams do not show all that you need to write. The few CASE tools I have seen that implement use cases through interaction diagrams, force the writer to hide the text of the steps behind a pop-up dialog box attached to the interaction arrows. This make the use case impractical to scan - the reader has to double click on each arrow to see what is hidden behind it. In the "bake offs" I have held, the use case writers and readers uniformly chose no tool support and simple word processing documents over CASE tool support in diagram form.

We must concede that these comments are relevant in our transformation. The use cases style that we are aiming at transforming is very simplified sentence structure. It is meant at representing use case steps such as *The user enters his information* and not like:

> ***System*** *extracts from the web request any and all user and navigation information, adds it to the logged information, and starts from some advanced point in the question / answer series.*

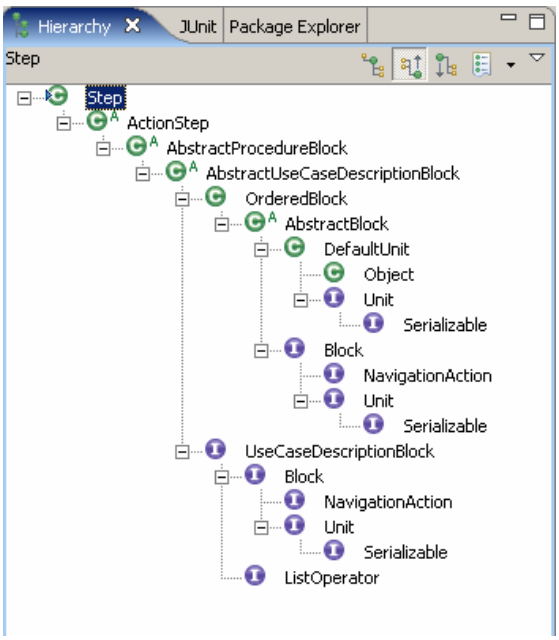
Using a restrictive style has the advantage of being automatically translatable and the disadvantage of not being as natural or expressive. Since our work did not concentrate on what you should express in use cases nor how you should express them, *Writing Effective Use Cases* wasn't our main reference. Our goal is simply to translate use cases into UCMs, we concentrated on articles by Vladimir Mencl and Stéphane Somé. Vladimir Mencl authored of a series of articles explaining how he translated textual use cases into pro-cases, a formal (textual) notation used by his research group. I presented one of his articles as my paper presentation this semester, *Deriving Behavior Specifications From Textual Use Cases* [6]. It distinguishes between different use case steps using NLP. His article is a good introduction to the NLP involved at the lower level, but since his pro-case notation does not support inclusions, other sources were needed in order to transform a set of interrelated use cases (such as in Figure 2) into a set of UCMs. Furthermore, he does not try to parse conditions (which are used in alternatives, for example). For these concepts, we referred to the different articles written by Stéphane Somé about UCEd and its user manual [10].

Converting use cases into another notation is not any groundbreaking material, but no tools exist to do it automatically. Since the concepts found in use cases are fairly similar to those found in UCMs, others have also presented general transformations. However, the material that we were able to collect presents it generally, saying that alternatives can either be converted into or-forks or into stubs.

Methodology

This prototype has been designed as a plugin to jUCMNav. In order to avoid having to re-implement the code that loads a UCEd file to parse its content and build an object-oriented structure to represent a UCEd file, we opted for a very simple solution, which is good enough for our prototype. We are basically including the UCEd 1.5.1 jar file in our build path, using its internal code to load an existing file and extracting the project model instance from UCEd. We traverse this model to build a URN file in jUCMNav. This task is done as an extension to the "import" extension point defined in jUCMNav. Once loaded, we perform an automatic layout of all the UCMs that were generated using the auto layout mechanism already present in jUCMNav, built on top of Graphviz dot. This allows us to delay any infrastructure related problems to later and concentrate on the real issue: transforming use cases into use case maps.

To accomplish this, we relied on two sub-sections of the UCEd project model. First, obviously, the actual use cases being described. Second, the domain model which is extracted from the Use Cases by UCEd. We added the constraint that, in order to be able to import a use case model, it must have been validated against its UCEd domain. UCEd includes a domain extractor module that generates the domain from the Use Cases, but this is not a headless process. The NLP used in UCEd is far from perfect and relies on the fact that the system analyst will confirm its decisions.



Since there exists no readily available meta-model for UCEd, a few reverse-engineering tools were used to understand its structure. Furthermore, we reiterate that we are only building a prototype. Hence, it is obvious that our transformation algorithm is pretty ad-hoc and has not been validated by the UCEd authors. Our goal is to have them validate the transformations explained in this report to further refine the conversion utility. From the limited prior knowledge we had about UCEd and using its user guide, we concentrated on a few features that we considered important. We did not use the generated state machines or scenarios defined in UCEd.

Figure 4: Use Case Step Ancestors and interfaces

Before proceeding with what is intended to be covered in our translation, let us restate what a use case step can be, as defined in [6].

- An operation received by the system under description from an actor
- An operation sent by the system to an actor
- An internal action executed by the system

Furthermore, a few special actions exist to manage the flow of a scenario such as a use case termination or a redirection.

The features we have decided to concentrate are listed below. Their UCM counterparts will be discussed in more detail in the Result Description section.

- **Actors**: The domain model defines the different entities that can be used in use cases. UCEd calls these *Concepts*. There are *System Concepts* and regular concepts. The former represents the system and the latter, actors. UCEd refines these into sub-concepts, sub-components and instances.
- **Primary successful scenario**: This is a sequence of interactions (sequence of use case steps) between a user and the system that usually occurs.
- **Alternatives**: After a certain step in the use case, if a condition is evaluated to be true, the scenario branches off to this alternative.
- **Redirection actions**: A certain step can redirect to another (*Goto step 3*); this is most commonly used in alternatives to return to the main scenario. Note that UCEd does not allow redirections into alternatives, which is a good thing if we want to keep our use cases readable.
- **Use case inclusion**: One use case can include another. The UCEd project file contains multiple use cases and defines their relationships.
- **Use case extensions**: One can define an extension point in a use case. Optionally, the system analyst can define extensions that will add behaviour to the use case containing the extension point.
- **Various conditions**: Primary successful scenarios have pre and post conditions, alternatives have conditions, use case extensions can also be conditional and a use case step can be of the type *IF User Password is Invalid THEN System displays error message*.

What we chose to ignore or only implement partially:

- **Time constraints**: UCEd allows conditions to contain time constraints such as *BEFORE 60s User accepts the terms and conditions.*
- **Any * alternatives**: Alternatives can be defined as available after a specific step or after any step.
- **Operation effects**: Operation effects are changes in the use case state when certain operations are performed. Operation are defined (in the domain model) on different Concepts. (Added/withdrawn conditions).

Data

The tool was applied to various use cases built using UCEd, targeting certain specific features (listed above) and tested manually. We also imported the sample UCEd projects that are distributed with the tool. Since this is only a

prototype, we did not invest effort into producing real-world examples. What could be done is to implement the use cases that are distributed in *Writing Effective Use Cases* and verify that they are correctly interpreted by our conversion utility. The source data presented in this report do not necessarily make sense; they are simple examples to describe the transformations.

Parameters

Our conversion utility does not have any configurable parameters. Our results are therefore easily reproducible. The UCMs that are produced and displayed in this report are manually edited versions of the auto-layout's output. Since the auto-layout mechanism is far from perfect, especially concerning label placement, some adjustments were made to their presentation. However, no new UCM elements were added, other than empty points which have no semantic meaning.

The only aspect which might make it more difficult to reproduce our results is that one must have some knowledge of how UCEd works in order to properly use the domain extractor. If the extracted domain is not the same the one we extracted, the results will be different.

Evaluation Methodology

Our results were manually evaluated by Jason Kealey. Where the translation was not obvious, Gunter Mussbacher and Daniel Amyot gave their opinions.

The main goal of this report is to illustrate the different elements that are transformed in order to facilitate discussion and validation by various students and professors who either have deep knowledge of use cases and UCMs or know about the features and limitations of UCEd. The primary evaluators that we foresee are Gunter Mussbacher, Daniel Amyot and Stéphane Somé. As we implemented most of jUCMNav, we know what its features limitations are. We admit, this evaluation methodology is very ad-hoc, but is, we believe, sufficient for a course project.

Result Description

This section contains a summary of our results. We will present one or more use cases, the generated UCM and a discussion of issues present in this translation. Note that we will not present the domain model unless necessary. Furthermore, I reaffirm that the examples presented here don't necessarily make much sense; they are toy examples in a sense. One important thing to remember: the key to being able to automate the conversion is to use simple English, namely the Subject-Verb-Direct Object-Preposition-Indirect object (SVDPI) pattern when writing use case steps. Note that the discussions often include limitations, relevance and future work.

1) Actors

Input:

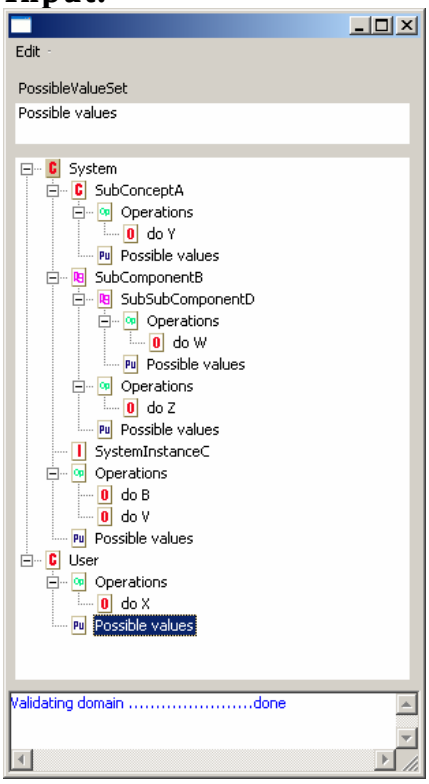


Figure 5: UCEd Domain Model

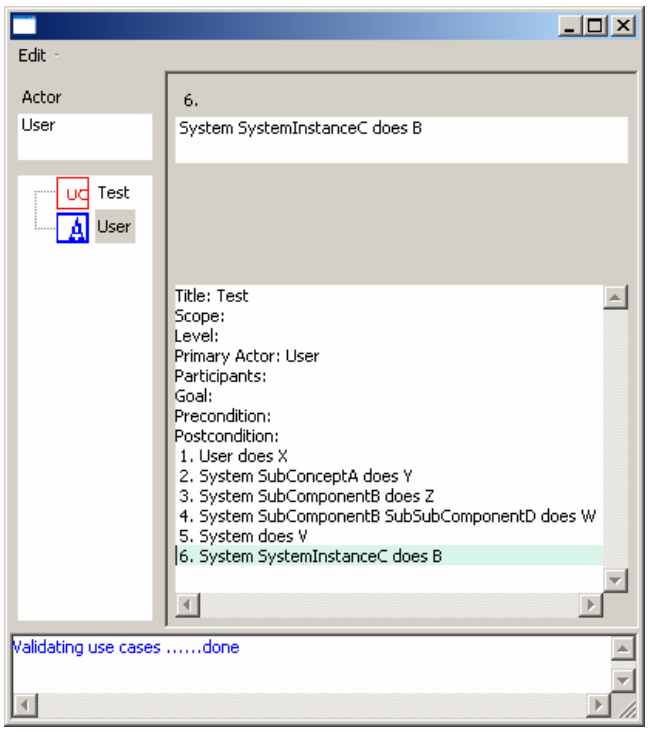


Figure 6: UCEd Use Case

Output:

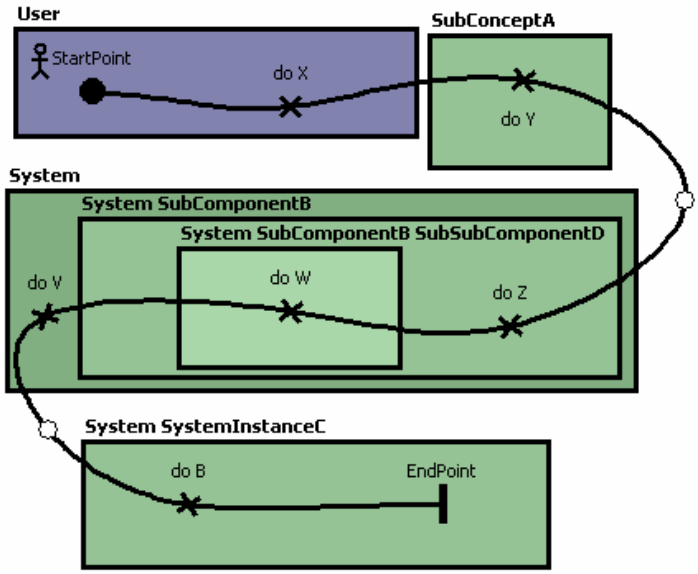


Figure 7: Generated UCM

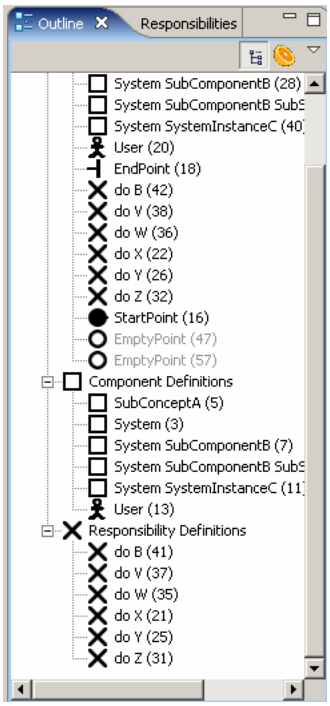


Figure 8: Generated URN elements

Discussion:

In the Subject-Verb-Direct Object-Preposition-Indirect object (SVDPI) pattern that a use case step should follow, Actors represent the subject. But how do we distinguish between Actors and the system under description (SuD)? UCEd lets the user perform this task manually in the domain extractor.

We have lots to discuss about actors since the concepts are rather confusing in UCEd. UCEd supports many different types of concepts. First, we have *System Concepts* and regular concepts. Both of these are related to UCM component definitions. We decided that to keep things simple, system concepts would be mapped to components of type *Team*, which are the default component type. Regular concepts, however, would be mapped to components of type *Actor*.

Simplified version of what follows:

1) **Every entity in the UCEd domain tree is mapped a UCM component definition.**
2) **Every instance of a subject in the SVDPI pattern of a use case step is mapped to a UCM component reference.**

If you're interested in details, read the following; otherwise skip to 2).

In UCEd, both types of concepts can have sub-concepts, sub-components and concept instances. Sub-components can only have other sub-components as children. Only concepts and sub-concepts can have concept instances as children.

For reasons unknown, system concepts are only defined as top level entities in the domain model in UCEd. Therefore, when traversing the different concepts in the domain model, the top-most parent is obtained and its type is verified in order to determine if the component definition to be created is an Actor or Team.

You should now be confused. Don't be ashamed, we are too! ☺ Therefore, we decided to simplify everything and map all concepts, system concepts, concept instances, sub-concepts and sub-components to distinct component definitions in UCMs.

We did not make this distinction, but it is useful to remark that up to now, we mentioned only component definitions which are entities that can have multiple component references associated with them in UCMs. A component reference is contained in a UCM and is related to its definition. Component references are the actual visual representation of component definitions (they are boxes) and they can have UCM elements bound to them (included in them in the diagram above). Therefore, we added another feature: given two UCM component references in a map related to two UCEd elements, if is a sub-component of the other, bind (include) the sub-component to its parent.

We add component references to a map only if a use case step has it as a subject. Otherwise, we would be adding many empty/useless components in every map. Still, even if no use case steps reference a certain concept, a corresponding component definition will still be created.

The definitions are named using the *getFullName()* method defined on concepts in UCEd. UCEd adds a constraint that if two concepts share the same parent, their names must be unique. Therefore, getFullName() recursively prefixes the parent's name to ensure global name uniqueness. This is great because UCMs component definitions must have a unique name. However, for reasons unknown, SubConceptA is not prefixed with System by *getFullName()*. This could introduce problems if naming conflicts occur during the conversion.

In conclusion, UCEd defines more information in its domain model than what is needed to translate to UCMs. This could be problematic if the inverse transformation was applied, especially because of the odd behaviour of sub-concepts.

2) Primary successful scenario

Figure 5 to 8 are sufficient to describe the transformation of the primary successful scenario. Simply put, every regular use case step is mapped onto a responsibility reference (an X) on a single path, flowing through component references. As mentioned previously, the path elements are bound to the component reference corresponding to the subject of the use case step.

The verb and direct object of the use case step are partially lemmatized by the UCEd domain extractor and added as possible operations of a particular concept. The UCM responsibility is named after the operation name in the UCEd domain.

Contrary to what is done for component definitions, we do not create responsibility definitions for operations in the domain model which do not occur in use case steps. We are doing this because we do not transfer any semantics associated with responsibility definitions (see 10: Operation effects) and because we want to keep things simple. It would not be difficult to make this mapping more uniform with the transformation applied to concepts.

The indirect object of the SVDPI pattern is not mapped to the responsibility name, it remains unused. In a UCM, we hide the underlying communication mechanisms. Therefore, for example, if a use case step said *User sends information to System*, we don't have anything visual to do with the indirect object (System). However, we could add it to the responsibility name to make it more explicit.

UCEd constrains that a concept's operations must be uniquely named. jUCMNav constrains that responsibility definitions have unique names (globally).

Therefore, we should prefix the responsibility name with its concept's full name, but this would hinder readability. Again, since no semantics are currently transferred, this is not a problem: we can re-use the same definition if different concepts have operations of the same name. However, this would not be possible of operation effects were implemented.

If responsibility definitions were associated to component definitions, this would be possible.

3) Alternatives

Input:

```
Title: General Scenario
1. User Jason clicks import button
2. jUCMNav GUI displays file list
3. User Jason selects file
4. UCEd loads Use Case
5. jUCMNav ImportExport UC2UCM converts UC into UCM
6. jUCMNav loads UCM
7. jUCMNav GUI displays UCM
3. a. User file is not found
3. a. 1. Goto step 2
4. a. UCEd Use Case is not valid
4. a. 1. jUCMNav GUI displays error message
```

Figure 9: Use Case illustrating alternatives

Output:

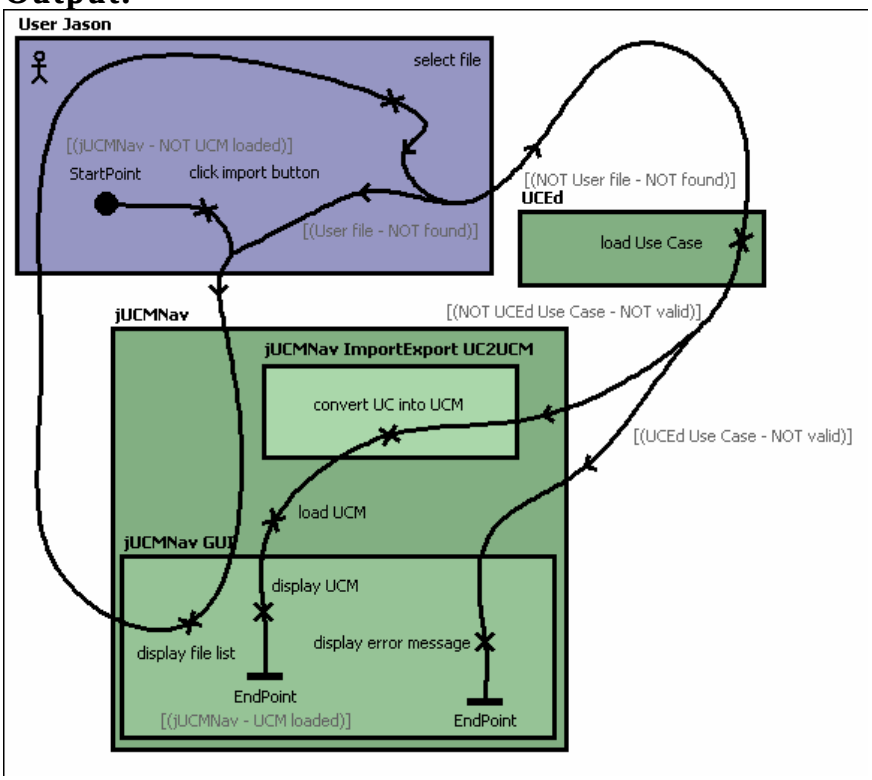


Figure 10: Generated UCM, with alternatives

Discussion:

Obviously, use case alternatives (3.a. and 4.a) are mapped to or-forks in UCMs. The branches leaving an or-fork all have conditions. Conditions in jUCMNav are displayed in gray, surrounded by square brackets. We will cover the pre-conditions and post-conditions later (close to the start point and one of the end points). Note that if multiple alternatives to the same use case step occur, an or-fork with more branches will be produced.

jUCMNav currently does not support scenarios and its use of conditions is simply visual for the time being. Therefore, we are simply taking the conditions and using their String representation in our UCMs. This also explains why double negation such as "Not user file is not found". We could have used UCEd's internal mechanism to eliminate double negation but there are implementation reasons on why we did not do this. Once jUCMNav supports scenarios, the double negation eliminations (and other manipulations) will be done automatically.

Because of the way the use cases are structured in UCEd, no natural language processing is required to discover alternatives. However, NLP is used to understand their conditions. UCEd defines possible values for all entities which can either be discrete and non-discrete.

To clarify this, let us present (slightly modified) excerpt of the UCEd documentation:

```
A simple condition must adhere to the following syntax.
[determinant] entity verb value

Possible determinants are: a, an, the.
Possible verbs are: is, isn't, is not, are, aren't, are not, has,
hasn't, has not, have, haven't, have not.

A value may be one of the possible values of the condition entity, or
specified as a general comparison.

For possible values, the sequence of words used to refer to the value
must be declared as a possible value of the entity in the domain model.

For general comparisons, the syntax for value specification must adhere
to the following: comparator value.

Possible comparators are: >, <, =, <=, >=, <>, greater than, less than,
equal to, different to, greater or equal to, less or equal to.

A complex condition is a negation, a conjunction or a disjunction of
conditions.

[(NO|NOT)] condition ((AND|OR) condition)*
```

We see that conditions are what we would expect from a propositional logic expression parser (See my CSI5110 work to be included in jUCMNav) applied to a certain domain. These conditions are more expressive than what is currently in jUCMNav's predecessor, UCMNav. BNF grammar is sufficient to extract the propositional logic part of the equation. However, I feel more NLP is needed to better understand the values. In my tests, I wrote conditions such as "User is a convicted felon" but the above grammar did not like this, since there are issues with having determinants in the value when using the domain extractor. I had to end up using "User is convicted felon" for it to work. I believe basic NLP parsing here would allow system analysts to express concepts more naturally. UCEd does not care what verb is used; that is why when we output their textual representation in jUCMNav we obtain a dash (-) instead of a verb, because any of the valid verbs could have been used.

4) Redirection actions

The previous example also includes a GOTO action. These are converted into or-joins. When redirecting to a certain step, add an or-join between this step and its predecessor and merge the current path with the given path.

In UCEd, one can write a redirection action only to *previous* steps in the primary successful scenario; therefore, one cannot redirect into an alternative. We kept this behaviour in jUCMNav partially because of the limitation in UCEd and (we admit) partially by laziness. Redirecting to a future step implies that we are creating an or-join before that future step but since we are evaluating the steps one after the other, that future step does not exist (yet) in the generated UCM. Obviously, one needs to do a two-pass implementation to be able to jump to a later step, but since this is only a prototype, we kept things simple. However, note that our implementation allows a redirection in an alternative to jump to any step (step 3.a.1 could have jumped to step 7 if desired).

Another issue brought up by redirection actions are implicit loops. The UCM notation only allows certain types of loops which can abstractly be defined as loops that are not infinite. As I had the fun job of implementing a checker in jUCMNav that verifies if a certain modification is allowed; that is if it does not create infinite loops. As mentioned previously, jUCMNav does not support scenarios nor any semantics on conditions. Therefore, the types of checks that are done imply verifying that when modifying the diagram, the next step will not create any loops that do not have any exit points. For example, take the following use case:

1. User clicks on import button.
2. System displays import window.
3. GOTO Step 1.

[Disregard the fact that the above use case makes no sense]. This would create a UCM that has one start point but no end points because the last step is a redirection back onto the same path. This is not allowed.

However, consider the following:

1. User clicks on import button.
2. System displays import window.
3. GOTO Step 1.

1. a. System Display is import window

1. a. 1. System displays error message

This would create an or-fork after step 1, allowing the loop to terminate. Therefore, a valid UCM can be drawn. Since we aren't evaluating the conditions, step 1.a might be a contradiction (never true) and the system would loop infinitely, but that is currently out of jUCMNav's scope.

Now that you are aware of the loop situation, be informed that when an illegal loop exists in a use case, because jUCMNav simply refuses to create the or-join, our import tool doesn't create one either (but it doesn't inform the user of this).

5) Use case inclusion

Input:

```
Title: General Scenario
Precondition: jUCMNav is not UCM loaded
Postcondition: jUCMNav is UCM loaded
1. User Jason clicks import button
2. jUCMNav GUI displays file list
3. User Jason selects file
4. include Load Use Case
5. jUCMNav ImportExport UC2UCM converts UC into UCM
6. jUCMNav loads UCM
7. jUCMNav GUI displays UCM

Title: Load Use Case
Precondition: UCEd ProjectModel is NOT loaded
Postcondition: UCEd ProjectModel is loaded AND UCEd
ProjectModel is valid
1. UCEd loads Use Case
2. IF UCEd Use Case is not valid THEN jUCMNav GUI displays
error message
```

Figure 11: Use case with inclusion, inline alternative and pre/post conditions.

Output:

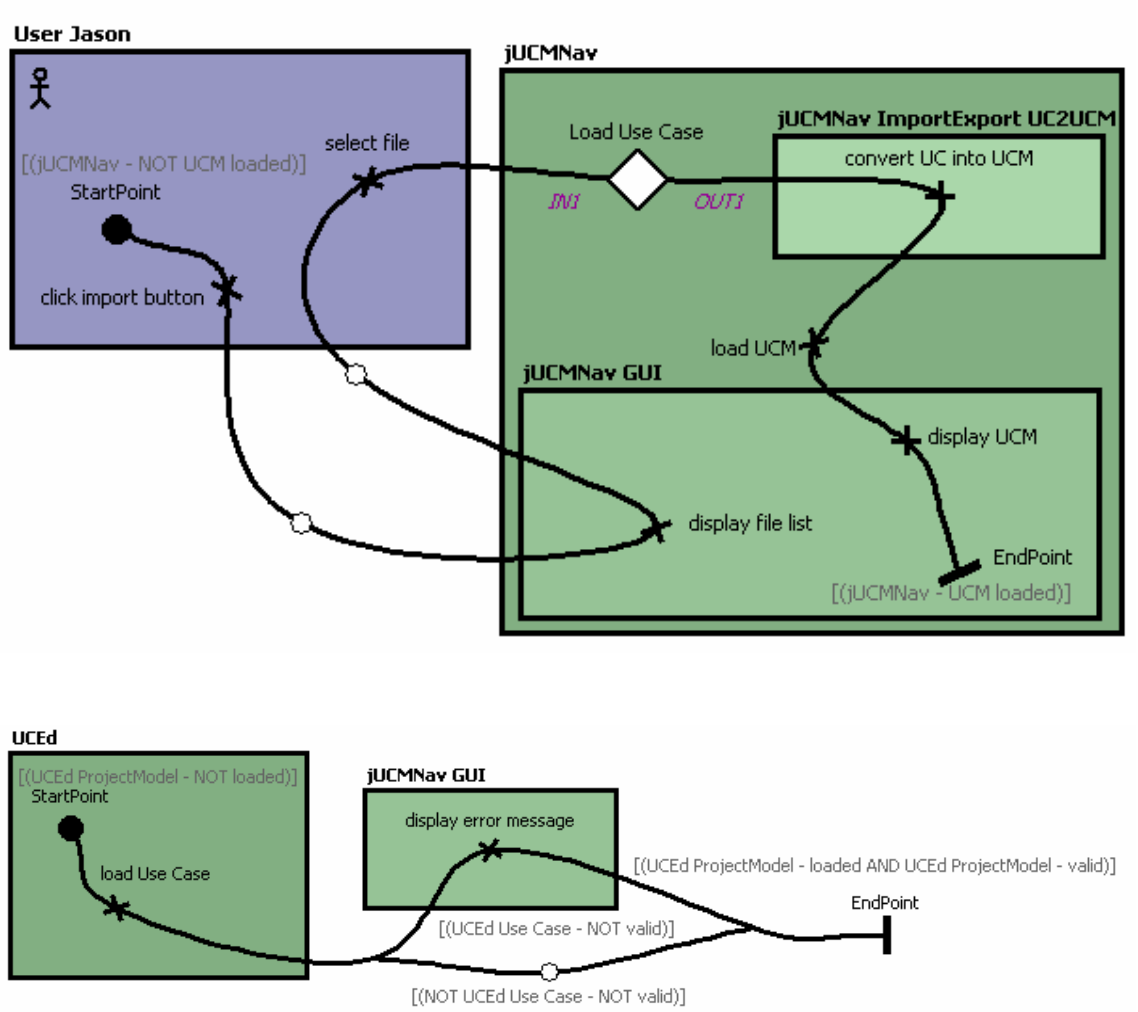


Figure 12: General scenario UCM (top) and Load Use Case UCM (bottom)

Discussion:

Not much NLP is involved here either, since we are simply specifying the include keyword followed by the name of another use case. An inclusion step is converted into a static stub (rhombus) which has a single plugin, the included UCM. Those familiar with UCM semantics know that stubs offer bindings which link incoming/outgoing paths with the start/end points of the plugin. Since our generated UCMs can have more than one start/end point, I have delayed the creation of these bindings because further study is required. The infrastructure is in place for the bindings to be created, and the primary scenario's end point is known and we can assume that the primary scenario's start point is the start point with the lowest (automatically-incremented) id in the map. However, there are possibly currently unknown issues relating to creating these bindings.

As for the inline if present in the plugin and the pre/post conditions, we will return to these in 7) Various conditions.

6) Use case extensions

Input:

```
Title: General Scenario
1. User Jason clicks import button
2. User Jason selects file
3. include Load Use Case
4. jUCMNav ImportExport UC2UCM converts UC into UCM
5. jUCMNav loads UCM
ExtensionPoint==> loaded
6. jUCMNav GUI displays UCM

Title: Logging
PART 1. At Extension Point loaded
1. jUCMNav logs usage
```

Figure 13: Use case with extension point and extension.

Output:

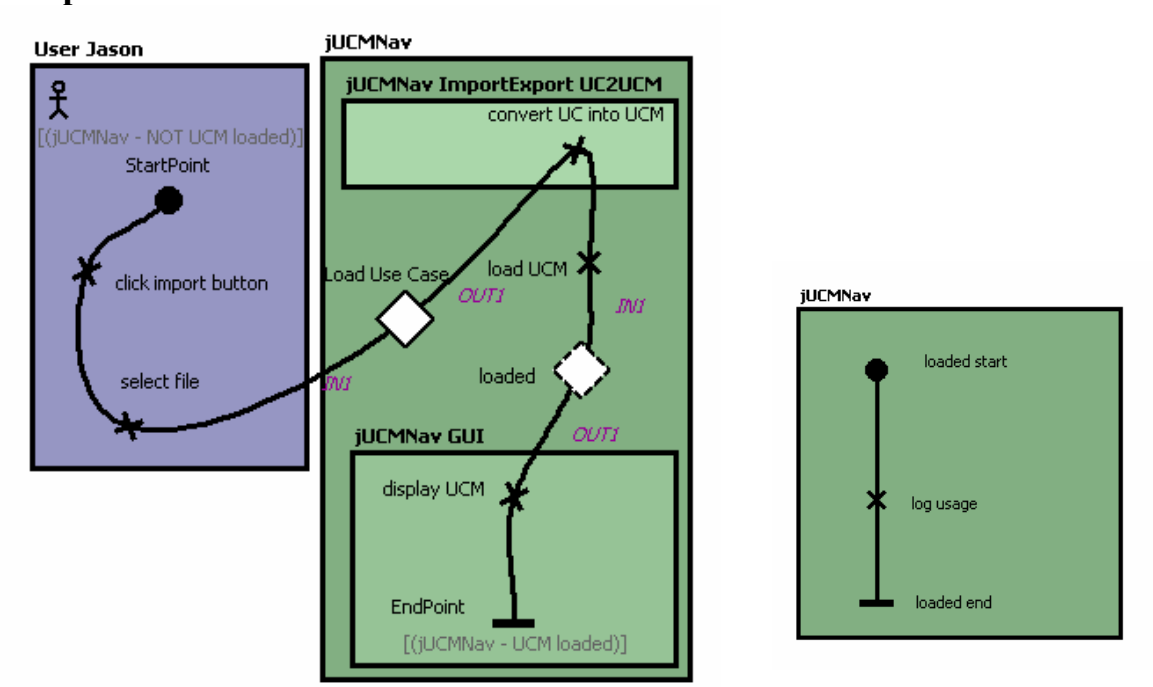


Figure 14: UCM with extension point (left) and its extension (right)

Discussion:

Here also, NLP is not of much use. An extension point is mapped into a dynamic-stub (dotted rhombus) which is named after the extension point name. The extension use cases in UCEd are treated specially. One defines an extension use case as being an extension of one or more specific use cases. An extension use case does not have certain properties that regular use cases do because they are inherited from the extended use case. Extension use cases can have multiple parts. Figures 13 and 14 show an extension with only one part. If multiple parts

were defined, multiple paths would be created in the corresponding map. The primary start and end points for these parts are named according to the extension point name, to help distinguish the different paths.

These start/end points should be used for plug-in bindings, but this leads me to discuss a shortcut I took while implementing this transformation.

Dynamic-stubs in Use Case Maps can contain one or more plugins. Each different plugin has a binding condition which, when evaluated to true, allows scenarios to determine which plugin should be traversed. One and only one of these conditions should evaluate to true for the UCM to be well defined.

However, here we are definition an extension point which could have no related extensions or that could have many extensions which must ALL be executed during the scenario traversal. To replicate this behaviour, the stub's plugin should be a new map which would contain a path that includes all the extensions. But, should these extensions be executed in parallel or sequentially? If sequentially, in what order? Is the order important? If not, how is this expressed in a UCM? These questions need answers.

For this project, I simply bound all the extensions to the stub representing the extension point and define the rest as future work. In my opinion, the semantics of UCM dynamic stubs should be enhanced to better express these concepts.

7) Various conditions

We've already discussed conditions in alternatives, but I have a few more comments to make concerning these, for future reference.

Figures 11 and 12 feature inline conditions and pre/post conditions. To keep a long story short, the preconditions are associated with the start point and the postconditions are associated with the end point. Inline conditions are IF condition THEN statements. There is no "ELSE" clause. Writing Effective Use Cases discourages the use of inline conditions, but since they are supported by UCEd, we included them in our traversal.

The semantic difference between alternatives and inline conditions is that inline conditions join back to the primary scenario after their action while alternatives fork away.

Furthermore, it is not mentioned extension in the previous section, but extensions can have extension conditions, meaning that behaviour is added to an extension point if and only if a certain condition is true. This condition is mapped to the plugin selection condition in the dynamic stub.

8) Time constraints

Input:

Title: General Scenario
Primary Actor: User Jason
1. User Jason clicks import button
2. User Jason selects file
3. BEFORE 60 sec jUCMNav ImportExport UC2UCM converts UC into UCM
4. jUCMNav loads UCM
5. jUCMNav GUI displays UCM
2. a. AFTER 60 sec
2. a. 1. jUCMNav GUI displays error message

Figure 15: Use case with time constraints

Output:

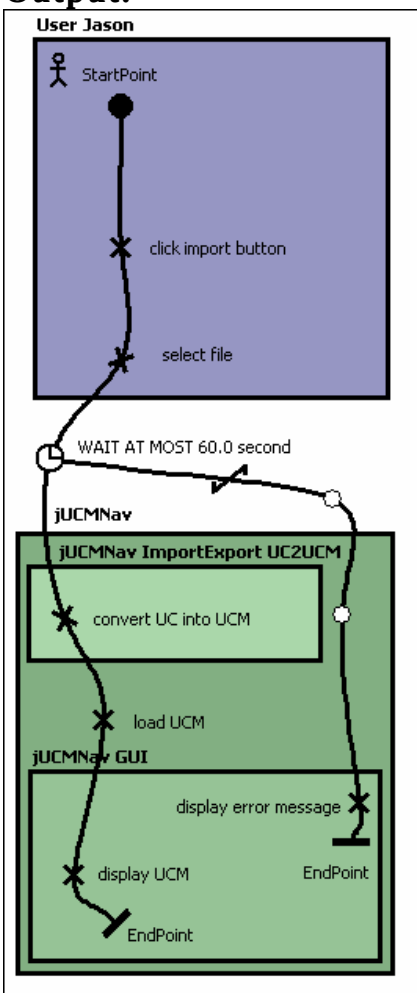


Figure 16: Use case map with a timer.

Discussion:

Timing conditions are one of the features we were most anxious as transferring as they presented technical challenges, but after experimenting with them a bit, we decided to keep them out of this project and implement them as future work.

In UCEd, use case steps can define that a certain step will occur BEFORE or AFTER a certain interval of time. Furthermore, alternative conditions can also use timing constraints, combining BEFORE, AFTER and regular conditions as defined previously.

Use Case Maps include a Timer construct which is a waiting place where the scenario is paused until another path activates the timer (the other path's end point is connected to the timer). The timer also specifies a timeout time after which it stops waiting and the original scenario goes down the timeout path (indicated with lines over the path). However, jUCMNav's meta-model currently doesn't support any way to define that actual timeout duration. And even if it did, representing before and after implies creating secondary paths that activate the timer as described by the BEFORE and AFTER conditions in UCEd. This is far from being visually intuitive for those who are not familiar with UCMs.

Therefore, we decided to drop this feature and simply add a timer, give it an expressive name such as shown in Figure 16 (WAIT AT MOST 60 seconds) . The Figure 15 use case step 3 does not declare what happens if the time constraints are not respected therefore what adds the timeout path to the timer is the alternative.

We also "cheated" our way through this feature. The alternative defines the timeout path because it is the logical opposite of the timer's timeout duration. However, we have no means of manipulating these conditions nor regular boolean expressions that can be added to the alternative. (An alternative could be taken if "BEFORE 60 s and AFTER 10s AND User Jason is convicted felon"). For this prototype, we decided not to spend any time reflecting on this situation and push it back to future work. We simply define that if an alternative exists on a use case step that uses a timer, the alternative will use the timeout path. If many of these alternatives exist, the timeout path will be forked.

We think that the issues brought up by this translation will provoke changes to the UCM notation so that these ideas are expressed more easily and the mapping becomes closer to what a human would intuitively write in use cases. However, if one wishes to change the semantics of timers, the new notation should not bring up non-deterministic decisions when the scenario traversal mechanism hits a timer.

*9) Any * alternatives*

Input:

```
Title: New
1. System displays welcome screen
2. User Jason clicks start
3. System displays information
* 1. System is quit requested
* 1. a. System displays welcome screen
```

Figure 17: Use case with any * extension.

Output:

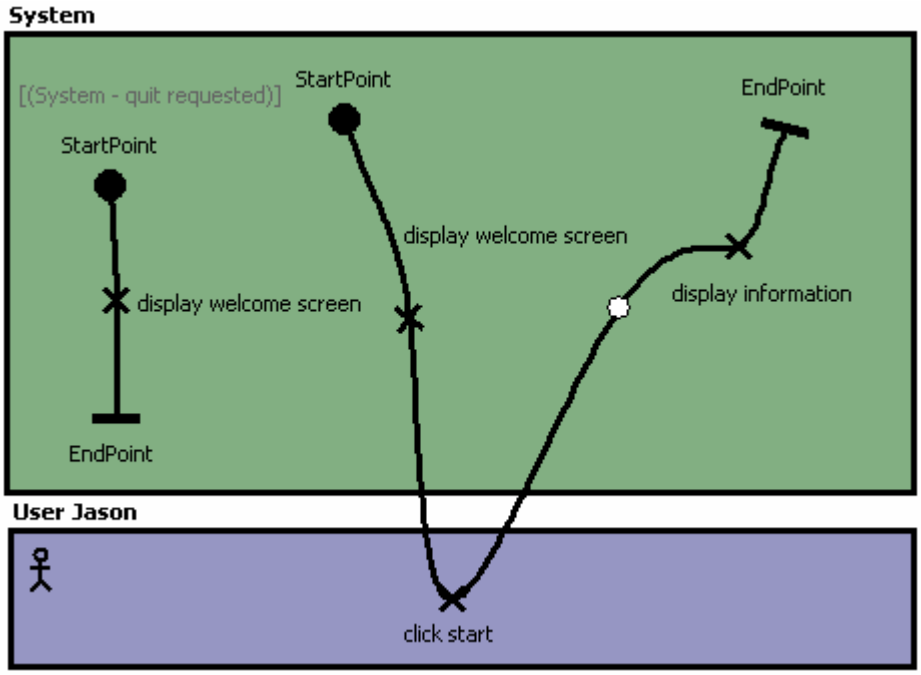


Figure 18: UCM with simple mapping for any * extension.

Discussion:

The alternatives discussed in 3) Alternatives can happen after a particular step but use cases defined in UCEd can also have Any * alternatives, which are defined to be executable after any step. The condition should be evaluated after each step. This is not easily expressible in the UCM notation without cluttering the graphic unnecessarily. Therefore, we decided to simply include the steps as separate paths in the same map having as precondition the alternative's condition. When discussing this matter with Daniel Amyot, we were informed that the UCM exception construct would help facilitate expressing this scenario, but we are leaving this as future work, since this construct is not currently implemented in jUCMNav.

10) Operation effects

UCEd allows operations to have effects on their concept's state. Using operation effects, one can perform simulations on use cases because operations now have an impact on which conditions evaluate to true and which evaluate to false. In UCM, this would be equivalent to defining the behaviour of a responsibility. Since scenarios are not yet implemented in jUCMNav, it was impossible to transfer this functionality and thus our responsibilities have no associated semantics.

Extracting operation effects from textual sentences using NLP techniques should be studied because these are not a part of the actual use cases, they are a part of the domain model.

Note that the previously mentioned issue concerning responsibility naming would come into play if operation effects were implemented.

Results Discussion

We believe we achieved our goal of implementing a tool for generating Use Case Maps from textual use cases. Many issues are mentioned in this report and should be addressed. We have discovered that use cases (at least the concepts implemented in UCEd) present features that don't easily map to UCMs. We were not expecting this, but we are happy to have gone through with the implementation of the tool. Taking this initiative certainly produced interesting work and knowing the differences between the two notations beforehand might have discouraged us from undertaking this project.

The main question we first need answered is: Should the UCM concepts be refined to be more similar to widely adopted use case concepts or should they remain as they are, as this gives UCMs their uniqueness?

Our project presentation presents table comparing Mencl's article with UCEd and jUCMNav/UCMs. Although we stated that we would explain this table in the report, seeing that it is already too lengthy, we will not discuss it here. If you are interested in more details, this can be discussed in person. The general aspects are all covered in this report.

Relevance

Previously in this report, we stated that our goal was to provide a method to help leverage existing use cases when first introduced to UCMs. In its current state our prototype does not achieve this as it relies on UCEd as its sole source for use cases. We need to be able to import use cases written in popular tools. Actually, UCEd needs to be able to import these (or from a free-form text document). From that point on, the use cases can be refined and the domain extracted, allowing for our conversion utility to be used.

Future Work

This report already documents many features that are not fully implemented and we do not wish to re-iterate them here. I simply want to add that UCEd should support scenario termination and extension abortion constructs (such as Use Case terminates), as defined in Vladimir Mencl's article. Furthermore, it needs a major usability overhaul and its meta-model should be simplified. (Expressing it using a regular UML modeling tool and then generating it using EMF (like jUCMNav does) would be a great addition). Visual feedback should be given to use case writers when they are actually in the writing process. If the writer is informed of what the domain extractor understands from the use case step, time spent validating and correcting use cases will be drastically cut down.

The main limitation that we found while implementing our prototype is that jUCMNav does not currently support scenarios, as has been frequently stated in this report.

The inverse transformation would also be interesting to study in the future. Furthermore, I believe we could generate UML Use Case Diagrams from our Use Case Maps. Use Case Diagrams could help propagate the "big picture" of a UCM model very quickly.

Conclusion

In this report we presented a prototype conversion utility that generates Use Case Maps from textual use cases expressed in UCEd. The work done here brings up many issues that should be discussed to help determine the future of the Use Case Map notation.

Furthermore, it helps define what I will be doing for my thesis: major contributions to jUCMNav (scenarios) and integration with other requirements engineering tools, using both forward and reverse engineering techniques:

Tool support for round-trip requirements engineering

Jason Kealey
jkealey@shade.ca
http://corp.shade.ca

References

[1] D. Amyot, *Introduction to the User Requirements Notation: Learning by Example*, Computer Networks, 42(3), June 2003, 285-301.
Available at: http://www.usecasemaps.org/pub/ComNet03.pdf

[2] D. Amyot and G. Mussbacher, "On the Extension of UML with Use Case Maps Concepts", *2000, 3rd International Conference on the Unified Modeling Language*, York, UK, October 2000. LNCS 1939, 16-31
Available at: http://www.usecasemaps.org/pub/uml2000.pdf

[3] A. Cockburn, *Writing Effective Use Cases*, Addison-Wesley Professional, USA, January 2000, 270p.

[4] Fantechi, S. Gnesi, G. Lami, and A. Maccari, *Application of Linguistic Techniques for Use Case Analysis*, Requirements Engineering, 2003.
Available at: http://rep1.iei.pi.cnr.it/FMT/WEBPAPER/RE02-revfin.PDF

[5] J. Kealey, E. Tremblay, J.-P. Daigle, J. McManus, and O. Clift-Noël, "jUCMNav: une nouvelle plateforme ouverte pour l'édition et l'analyse de modèles UCM", *Actes du 5ième colloque sur les Nouvelles Technologies de la Répartition (NOTERE 2005)*, Gatineau, Canada, août 2005, 215-222.
Available at: http://www.site.uottawa.ca/~damyot/pub/notere05.pdf

[6] V. Mencl, "Deriving Behavior Specifications from Textual Use Cases", *Proceedings of Workshop on Intelligent Technologies for Software Engineering (WITSE04)*, Linz, Austria, September 2005, 331-341.
Available at: http://nenya.ms.mff.cuni.cz/publications/Mencl-DerivingBehSpec-WITSE04.pdf

[7] V. Mencl, *Use Cases: Behavior Assembly, Behavior Composition and Reasoning*, Ph.D. Thesis, June 2004.
Available at: http://nenya.ms.mff.cuni.cz/publications/mencl_phd.pdf

[8] F. Pasil and V. Mencl, "Getting "Whole Picture" Behavior in a Use Case Model", *Proceedings of IDPT 2003*, Austin, Texas, U.S.A., December 2003, 63-79.
Available at: http://nenya.ms.mff.cuni.cz/publications/PlasilMencl-IDPT2003.pdf

[9] D. Richards, K. Boettger,, and O. Aguilera, "A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices", *Proceedings of AI 2002*, Canberra, Australia, Dec 2-6, 2002 LNCS 2557 Springer 2002.
Available at: http://www.comp.mq.edu.au/~richards/papers/ai02.ps

[10] S. Somé, *Use Cases based requirements engineering with UCEd*, UCEd User Guide.
Available at: http://www.site.uottawa.ca/~ssome/publis/ucedGuide.pdf

[11] S. Somé, "Beyond Scenarios: Generating State Models from Use Cases", *ICSE 2002 Workshop Scenarios and state machines: models, algorithms, and tools*, May 2002.
Available at: http://www.site.uottawa.ca/~ssome/publis/ICSE02_Scenario_Workshop.pdf

[12] S. Somé, "An approach for the synthesis of State transition graphs from Use Cases", *2003 International Conference on Software Engineering Research and Practice (SERP'03)*, June 2003.
Available at: http://www.site.uottawa.ca/~ssome/publis/ssomeSERP03.pdf