UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

2006.2

AUTOMATIC ENGLISH REQUIREMENTS GENERATION FROM CSP MODELS

TRABALHO DE GRADUAÇÃO

Aluna – Glaucia Boudoux Peres (gbp@cin.ufpe.br)

Orientador – Alexandre Cabral Mota (acm@cin.ufpe.br)

Recife, 03 de Abril de 2007

UNIVERSIDADE FEDERAL DE PERNAMBUCO

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CENTRO DE INFORMÁTICA

2006.2

AUTOMATIC ENGLISH REQUIREMENTS GENERATION FROM CSP MODELS

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aluna – Glaucia Boudoux Peres (gbp@cin.ufpe.br)

Orientador – Alexandre Cabral Mota (acm@cin.ufpe.br)

Assinaturas

Este Trabalho de Graduação é resultado dos esforços da aluna Glaucia Boudoux Peres, sob a orientação do professor Alexandre Cabral Mota, sob o título de "*Automatic English Requirements Generation from CSP Models*". Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

Glaucia Boudoux Peres

Alexandre Cabral Mota

To my parents

Acknowledgments

I want to thank everyone that, somehow, has been part of my life. This work is a result of dedication and support from those to whom I give my sincere thanks:

- First I want to thank God for being always by my side.
- I am vastly grateful to my parents, Gaspar and Conceição, to whom I dedicate this work. They have taught me how to live with responsibility and honesty. They are my *safe port* and it is because of them that I make an effort everyday to be a better person.
- I do thank my sister, “Mana”, and my brothers, Valmir and “Budu”, for the encouragement, comprehension, friendship, and attention they have always given to me.
- I especially thank Marcos for every moment we spend together, for all the wonderful talks we have about life and learning, for being always ready to help me, and for giving me so many advices.
- I also thank all my friends for being always there for me, for all the entertainments we have together, and for understanding my moments of absence.
- It would be unfair not thanking to some friends in particular, who have shared with me almost every moments of my academic life. I really want to thank “Thiaguinho” for the company and the nice talks about our fears and indecisions on the way to the University, and also for pushing me to do this work. I would not forget to thank “Gigil” for always treating me with affection, and for showing me friendship and comprehension. And I also want to thank “Guedes” for his great sense of humor. He always got to transform any tiresome work into a great laughter.
- And I thank my supervisor, Alexandre, for having trusted me throughout this work.

Resumo

Na maioria dos sistemas de software, requisitos estão sempre sendo modificados. Uma razão comum para isso é que a verdadeira natureza do problema, o qual o software está tentando resolver, somente emerge quando o projeto começa a ser desenvolvido. Normalmente, com uma mudança nos requisitos, o projeto do sistema e a implementação mudam também. Mediante isto, o resultado é que todo o sistema deve ser testado novamente. Como os casos de teste e a implementação tendem a representar a atualidade do projeto, é natural que a informações contidas neles sejam utilizadas para atualizar os documentos de requisitos.

Atualmente, existe uma forte tendência para adotar modelos formais a fim de representar os requisitos e os seus documentos associados. Modelar estes documentos com especificações formais e escrevê-los com uma linguagem natural controlada, para que se evite a introdução de ambigüidade e sentenças não-uniformes, é uma estratégia efetiva para garantir que nenhuma incerteza a respeito dos seus conteúdos esteja presente.

O objetivo principal do nosso trabalho é apresentar uma ferramenta que ajude na manutenção de documentos de requisitos, atualizados através de especificações formais em CSP correspondentes aos casos de teste.

Palavras-chave: CSP, Geração Automática de Casos de Uso e Linguagem Natural Controlada

Abstract

Requirements of most software systems are constantly being modified. A common reason for this is that the real nature of the problem the software is trying to solve only arises when the project begins to be developed. As requirements change, the system project and the implementation also must be changed. The whole system has to be tested again. It is natural that requirement documents become updated from test cases and implementation information, as test cases and implementation tend to represent the most recently changes made in the project.

Nowadays, the use of formal models is increasing in industry, particularly to represent requirements and their related documents. Modeling these documents with formal specifications and writing them in a controlled natural language, so that the introduction of ambiguous and non-uniform sentences is prevented, is an effective strategy to guarantee that their contents be consistent.

The main approach of our work is to present a tool that helps in the maintenance of requirements documents updated through formal specifications in CSP notation.

Keywords: Controlled Natural Language, CSP and Automatic Use Case Generation

Contents

List of Figures

1. Introduction

Requirements, for the majority of software systems, are constantly being modified. One of the reasons for this is that the systems are usually developed to deal with problems so complex, with so many related entities, that there is no definitive specification for the problem. The real nature of the problem only arises when a software solution begins to be developed. As the problem may not be entirely defined, the requirements of the system remain necessarily incomplete.

Another non-obvious reason for the frequent changes in requirements, but not less important, is the fact that most clients know the problem they want to solve, but not the technical solution for it. It results in requirements modifications due to not knowing – or not accepting – the solution chosen. The search for a final solution, which agrees with how the client views the problem and how he deals with it, brings changes.

Besides, after the final users become familiar with the new developed system new requirements can arise.

According to [Som03], changes in requirements happen because:

- The ones who pay for the system may impose requirements for its development because of organizational or budgetary reasons. These requirements may conflict with the requirements of the final users. This could happen due to the fact that, most of the time, who pays for the development, implantation and maintenance of the system is not one of its final users.
- A new hardware may have to be implemented and may be necessary to do an interface of the system with other systems. Besides, the priorities of the organization, where the system is being used, may modify, consequently bringing changes to the necessary system's support. New legislations and regulations also may be created and they have to be implemented by the system. The reason for all of this is that the company and the technical system environment may change, and it has to be reflected in the way the system behaves.
- Large systems usually have a varied community of users. This means that, different points of view of the system come with different users, in other words, different people see different solutions. Those points of view bring about different requirements and priorities that may be conflicting or contradictories. The final requirements of the system are obtained from the conciliation of the different users' points of view. It is evident that the balance of the support given to the different users need to be changed sometimes.

Throughout the software life cycle, requirements documents and their related documents, such as test artifacts, always need to be updated (maintained). The maintenance of these

documents is important because if an error occurs in one of them, the project can have a lot of re-working efforts (costs). Such costs may be even greater if an error is discovered during later phases of the development when the system is already in operation.

The cost to make a modification in a system, resulting from a requirement problem, is much greater than a modification during the design or code phases [Som03]. Tracking changes properly can save time and money, by identifying and fixing potential problems early in the development cycle [HT99]. A change in the requirements usually means that the system project and the implementation also have to be modified, and that the system has to be tested again. Consequently, the test artifacts also have to be updated with the new changes.

Nowadays, the use of formal models, which are an abstract way to specify computer systems to represent requirements and their related documents, is increasing in industry. Requirements need to be specially treated in order to produce high quality documents. These documents are the input to the formal specification activity and uncertainties should be avoided. Investing in good requirements specification methodologies is an effective way to reduce cost.

Once a project has precise documents, it is possible to create a formal specification in order to validate the system properties. The manual creation of formal models specification may not cover all requirements or may contain inconsistencies regarding them. Thus, the necessity of automatically generate requirements documents' formal models seems to be an efficient task to be accomplished. In order to automate the construction of formal models, the requirements documents should be simple, direct, unambiguous and uniform. For it happens, simple languages are used to describe them. These languages are called Controlled Natural Languages (CNL) [SLH03]. They contain a smaller and restricted grammar than the natural languages. Thus, they prevent the writer from introducing ambiguous and non-uniform sentences.

The work presented here is in agreement with the reality of software requirements maintenance, making it possible to keep their related documents continuously updated. The main approach of our work is not to hit the right requirements in the first moment, but to be ready to react when they change.

The main contributions of our work are as follows:

- A tool to translate CSP models, written in accordance to CSP$_M$ (Machine-readable CSP), into requirements documents written in CNL;
- A modification in the format of the CSP files proposed by [Cab06], which are our tool input files, so that it is possible to successfully transform them into requirements documents. This modification is necessary to recover some original documents specification information, which is lost after the requirements transformation into CSP models by [Cab06].

1.1. Objectives and Context

This work has been developed in the context of the CIn/BTC research project, which is sponsored by Motorola Inc. [Inc07] in cooperation with CIn/UFPE [CIUFPE07]. This cooperation started in 2003 and, initially, aimed at the creation of human resource specialists in the area of Software Testing. The CIn/BTC is divided in three areas, formal education and hands-on training, operation, and research. This work is related to two of the research team projects.

Figure 1.1, shows the research project initiatives that aim to improve the software development process through automation.

One of the researcher's projects [Sou06] was to develop a tool that automatically translate Test Cases, written in an English CNL with a fixed grammar [Lei06], into formal model specifications and their way back to Test Cases written in the same CNL.

Another researcher's project [Cab06] was to develop a tool that automatically translate Use Cases, written in the same English CNL as the previous project, into formal models specifications, making it possible to transform Use Cases into Test Cases, both written in the same CNL, and to maintain their documents constantly and automatically updated.

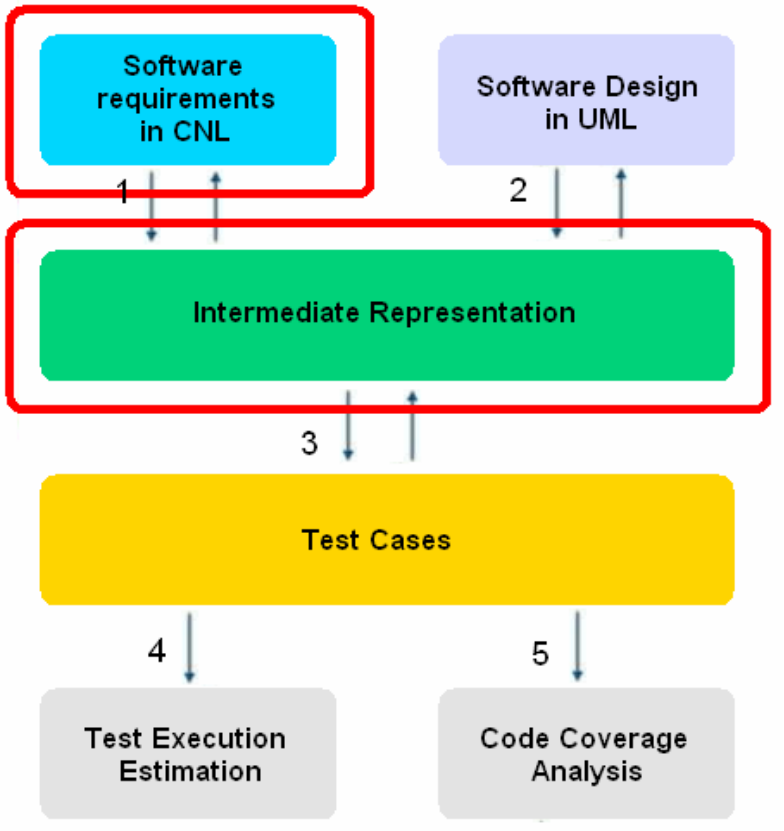

Figure 1.1 Research project initiatives overview

As the context of these projects is the research cooperation between CIn/UFPE and Motorola, related to mobile applications testing, the CNL used by them and the formal specification adopted reflects this domain. Because mobile applications may contain complex features, which include concurrent behavior and message exchanging, the formal specification chosen is the process algebra CSP [Ros97].

From our previous brief explanation about the projects [Sou06] and [Cab06], the translation from Test Cases to Use Cases was not yet implemented by a tool: the translation from the formal model to the use cases they represent written in CNL, emphasized in Figure 1.1.

In this context, the main goal of our project presented here is to develop a tool that automatically translates a use case document represented in the CSP process algebra into its corresponding use case document written in the same English CNL used by [Sou06] and [Cab06]. Such a tool must provide the translation of use cases documents, represented in the CSP notation, into their corresponding Microsoft Word 2003 [LLM04] documents, written in the English CNL.

1.2. Document Organization

Chapter 2 presents an overview of the CSP language, describing its basic concepts.

Chapter 3 introduces the English Controlled Natural Language with a fixed grammar used to represent the use cases steps.

Chapter 4 presents our main contribution: a tool to translate use case documents represented in the CSP notation into corresponding documents written in English CNL.

Chapter 5 discusses the results obtained after experiments using the proposed tool.

And finally, **Chapter 6** summarizes our contributions, contrasts the proposed solution with related work, and suggests topics for further research.

2. CSP Overview

CSP (Communicating Sequential Processes) may be defined as a formal language for describing patterns of interaction in concurrent systems [Ros97]. CSP allows the description of systems in terms of component processes that operate independently, and interact with each other through message-passing communication.

The language of CSP was designed for describing systems of interacting components, and it is supported by an underlying theory for reasoning about them [Sch99]. The conceptual framework taken by CSP is to consider components, or processes, as independent self-contained entities with particular interfaces through which they interact with their environment. This viewpoint is compositional, in the sense that if two processes are combined to form a larger system, that system is again a self-contained entity with a particular interface: a larger process.

The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from a few primitive elements.

In our approach, the CSP process algebra was the formalism adopted to express concurrence and parallelism between the components in an effective way.

The next sections will present basic elements of CSP.

2.1. Basic Concepts

Processes are the basic entities that capture a behavior. Each process can be defined by equations and, in general, a set of process is used to get modularity. Beyond denoting modules of a system, the name of a process can denote the state of a process.

The behavior of a CSP process is described in terms of events, which are immediate operations, like OPEN or CLOSE that may transmit information. A primitive process can be understood as a representation of a basic behavior. There are two primitive processes: STOP and SKIP. STOP is the process that doesn't communicate anything and it is used to describe the break of a system, as well as a deadlock situation. SKIP is the process that indicates that the execution was contained with success.

Since a process interacts with other processes only through its interface, the important information in the description of a process concerns its behavior on that interface.

The interface of a process is described as a set of *events*. An event describes a particular kind of atomic indivisible action that can be performed or suffered by the process. In describing a process, the first issue to be decided must be the set of events which the process can perform. This

set provides the framework for the description of the process. Each event must be identified by unique name. One event may occur many times in the process behavior. Along with events, *datatype* are defined to structure the data transmitted between events. The set of datatypes and events, defined during the formal model specification of a specific domain application, is called an *alphabet*.

A *channel* has the same behavior of an event, but communicates a value. It can be an input or output channel. When an event is defined it is possible to determine if it will communicate some information or if it will only represent a specification phenomenon.

Processes communicate with each other through synchronization in their events. These communications could or not carry data, which means that a process can pass to or receive information from other processes. The *communication* in CSP means: Interaction, Observation and Synchronization. Interaction, because two or more processes interact through the communication; Observation, as we can only observe the behavior of the processes through its communication; and Synchronization, when two processes, that are executing in parallel, synchronize their executions through the communication.

2.2. Operators

There many CSP operators and we present in this section some of them. All the CSP operators can be found in [Hoa85] and in [Ros97].

2.2.1. Prefix

The prefix is the simpler operation involving a process. It defines a process engagement on an event and then the process behavior is like the suffixed process.

Let *x* be an event and *P* a process, the (*x* → *P)* represents the process that waits indefinitely by *x*, and then behaves like the process *P*. This operator can be used to model recursive processes.

2.2.2. Recursion

Recursion in CSP is the ability of a process to enter a loop behavior. The operator (→), presented in the previous subsection, can be used to model recursive processes. The behavior of the processes (*P* = *x* → *P)* is the indefinite repetition of the event *x.*

2.2.3. Sequential Composition

Processes execute when they are invoked, and it is possible that they continue to execute indefinitely, retaining control over execution throughout. It is also possible that control may pass to a second process, either because the first process reaches a particular point in its execution where it is ready to pass control, or because the second process demands it. The mechanism for transferring control from a terminated process to another process is sequential composition.

The sequential composition operator used is (;). The process (*P* = *A* ; *B*) initially behaves like *A*, and the like *B* when *A* is terminated successfully.

2.2.4. Choice

By means of prefixing and recursion it is possible to describe objects with a single possible stream of behavior. However, many objects allow their behavior to be influenced by interaction with the environment within which they are placed. If *x* and *y* are distinct events (*x* → P | *y* → *Q*) describes an object which initially engages in either of the events x or y. After the first event has occurred, the subsequent behavior of the object is described by P if the first event was x, or by Q if the first event was y. Since x and y are different events, the choice between P and Q is determined by the first event that actually occurs.

There are also the operators for *External Choice* and *Internal Choice*. The operator (□) belongs to External Choice. In this case, the environment controls the choice between the options of behavior. The process (*a* → *P* □ *b* → *Q*) tries to communicate the initial events *a* and *b*. If the environment accepts to communicate *a,* the process starts to behave like *P*. On the other hand, if the environment accepts to communicate *b*, the process starts to behave like *Q*. Due to the fact that the environment is the one who controls the choice between the behaviors, the operator (□) is also known as deterministic choice.

The operator (Π) belongs to Internal Choice. This operator is similar to the previous operator but denotes a process that behaves like either of the processes in a nondeterministic way without the knowledge of the external environment. The process (*x* → *P* Π *x* → *Q*) means that if *x* is an event from both *P* and *Q*, the choice between them is non-deterministically defined. The process behaves like *P* or *Q*, arbitrarily.

2.2.5. Parallel Composition

When two processes are put in concurrent execution, in general, the desire is that one interacts with other. The interactions can be viewed as events that require the simultaneous participation of both processes. Let *P* and *Q* be processes with the same alphabet, *P* || *Q* represents a process in which *P* and *Q* must be synchronized in all events. So an event *x* only occur when both processes are ready to accept it.

The process *P* [|*X*|] *Q* synchronizes *P* and *Q* in the event of the set *X*. *P* and *Q* can interact independently with the environment through the events outside the set *X*.

The process *P* ||| *Q* allows *P* and *Q* to execute concurrently without synchronization between them. Each event, offered to interleave of two processes, occur only in one of them. If both are ready to accept that event, the choice between the processes is non-deterministic.

3. Controlled Natural Language

This chapter introduces the Controlled Natural Language (CNL), which can be seen as a processable version of English and is used to write use case steps making it possible to accomplish validations and transformations.

The CNL grammar is basically a subset of the English grammar. Its sentences construction contains domain specific verbs, terms, and modifiers. The phrases construction is centered on the verb. Domain terms and modifiers are combined in order to take thematic roles around the verb [Fil76]. This strategy is detailed in [Lei06] where it has been used to translate test cases sentences into CSP constructions. The following sections describe the knowledge bases used to store these vocables involved in the definition of the CNL. In our approach, these knowledge bases are used by our tool (specified in Chapter 4) in order to achieve the translation from CSP constructions to use cases CNL sentences.

3.1. Lexicon

The Lexicon stores vocables that may appear in CNL sentences. Each vocable may be a *verb*, a *term*, or a *modifier*. Each one of these vocables will be described in the following subsections.

3.1.1. Verb

A verb is used to define an action accomplished by the user, to give a description of the system state and the system response, or to specify a message. Actions are described as imperative commands, such as a statement to the user or component to accomplish some operation. In the case of the user view, the verb usually acts as a command in the user action column sentences, which are operations that the user executes in order to obtain a system response.

```
<verb>
  <name></name>
  <third-person></third-person>
  <gerund></gerund>
  <past></past>
  <participle></participle>
</verb>
```

Figure 3.1 Verb definition

Figure 3.1 is the XML [RM01] structure used to define CNL verbs. The verb definition contains possible verb forms, such as present or past tense. The *name* tag contains the verb in the infinitive form. The infinitive form is used in present tense sentence constructions. The *third-person* tag defines the singular third person forma as it may vary. The *gerund* tag contains the gerund form of the verb (-ing). The *past* tag defines the past tense form of the verb and the *participle* tag its participle form. Figure 3.2 contains some examples of verb definitions.

```
<verb>
  <name>have</name>
  <third-person>has</third-person>
  <gerund>having</gerund>
  <past>had</past>
  <participle>had</participle>
</verb>
<verb>
  <name>accept</name>
  <third-person>accepts</third-person>
  <gerund>accepting</gerund>
  <past>accepted</past>
  <participle>accepted</participle>
</verb>
```

Figure 3.2 Verb definition examples

3.1.2. Term

A term is considered an element, or entity, from the application domain. It may be just a noun or a noun combined with adjectives or other nouns. It is seen as an application domain object that is manipulated somehow by user or by components throughout the use case execution. The Figure 3.3 is the XML [RM01] structure used to define CNL terms.

```
<term>
  <name></name>
  <plural></plural>
  <class></class>
  <model></model>
</term>
```

Figure 3.3 Term definition

The *name* tag is the term's name itself. It contains the singular form of the term. The *plural* tag contains the plural form of the term. The *class* tag defines the Ontology class it belongs to, which determines how the term is related to other terms. Finally, the *model* tag contains the CSP code representation of the term. This representation is used to define CSP datatype values. Figure 3.4 contains some examples of term definitions.

```
<term>
  <name>conversation history</name>
  <plural>conversation histories</plural>
  <class>list</class>
  <model>CONVERSATION_HISTORY</model>
</term>
<term>
  <name>voice mail</name>
  <plural/>
  <class>application</class>
  <model>VOICE_MAIL</model>
</term>
```

Figure 3.4 Term definition examples

As can be seen in Figure 3.4, the term *conversation history*, for instance, has the plural defined as *conversation histories* and belongs to the *list* class of the Ontology. Consequently *conversation history* is treated as a list by the verbs that refer to this class. Finally, its CSP code is defined as CONVERSATION_HISTORY.

3.1.3. Modifier

A modifier can be anything that qualifies a term, such as an adjective or an adverb. It may be even a noun, once nouns can detail terms characteristics. Figure 3.5 is the XML [RM01] structure used to define modifiers.

```
<modifier>
  <name></name>
  <position></position>
  <precedence></precedence>
  <number></number>
  <article></article>
  <model></model>
</modifier>
```

Figure 3.5 Modifier definition

Over again, the *name* tag specifies the name of the modifier. The *position* tag defines whether the modifier goes before or after the term. The *precedence* tag specifies the priority order among the modifiers. The *number* tag is used to determine whether the modifier agrees with a singular or a plural term. Then, the *article* tag defines whether the modifier can be preceded by a definite or an undefined article. Finally, the *model* tag contains the CSP code representation of the modifier, so it is used to define CSP datatype values.

```
<modifier>
  <name>some</name>
  <position>before</position>
  <precedence>0</precedence>
  <number>plural</number>
  <article>no</article>
  <model>SOME</model>
</modifier>
<modifier>
  <name>at most</int></name>
  <position>before</position>
  <precedence>0</precedence>
  <number>singular</number>
  <article>no</article>
  <model>AT_MOST.Int</model>
</modifier>
<modifier>
  <name>with</int></term></name>
  <position>after</position>
  <precedence>0</precedence>
  <number>singular</number>
  <article>no</article>
  <model>WITH_N_NOUN.Int.Item</model>
</modifier>
```

Figure 3.6 Modifier definition examples

Figure 3.6 illustrates three modifier definitions. The *with <int/> <term/>* modifier is used along with an integer and a term. To better understand this modifier, consider as an example the sentence *Create a message with 3 images.* The number *3* is the integer that goes after *with* and *images* is the other *term* required by the modifier construction.

3.2. Ontology

As mentioned before, each application domain has specific elements and entities. They are called terms and are grouped into classes according to their characteristics. These classes can also be related by inheritance. An Ontology defines the grammatical classes and their hierarchies.

```
<ontology>
    <class>
       <description>Generic Class</description>
       <name>Object</name>
       <code>object</code>
       <subclasses>
          <class>
             <description>Represents a generic value</description>
             <name>Value</name>
             <code>value</code>
             <subclasses>
                <class>
                   <description>Represents a state value, e. g.,
                               "enabled", "ON", "high". </description>
                   <name>State Value</name>
                   <code>state_value</code>
                   <subclasses />
                </class>
                <class>
                   <description>Represents a value used to fill a field,
                            e. g., "some chars", "uppercase characters".</description>
                   <name>Field Value</name>
                   <code>field_value</code>
                   <subclasses />
                </class>
                <class>
                   <description>Represents a location value, e. g., "valid destination".</description>
                   <name>Location Value</name>
                   <code>location_value</code>
                   <subclasses />
                </class>
                <class>
                   <description>Represents a color, e. g., "blue", "different color".</description>
                   <name>Color Value</name>
                   <code>color_value</code>
                   <subclasses />
                </class>
             </subclasses>
          </class>
...
```

Figure 3.7 Ontology definition example

Figure 3.7 illustrates a small fragment of the Ontology that defines the *Object*, *Value*, *State Value*, *Field Value*, *Location Value*, and *Color Value* classes. The *State Value*, *Field Value*, *Location Value*, and *Color Value* classes inherit the *Value* class, and the *Value* class inherits the *Object* class. In Figure 3.4, the term *conversation history* is a *list* due to the fact that it belongs to the *list* class of the Ontology. This class aims to restrict the way terms are combined with verbs to avoid inconsistent sentences in the use cases.

3.3. Case Frame

The case frame defines the relation between verbs and terms, more specifically it defines how ontology classes are combined with verb complement definitions. Each case frame determines how a verb can be used to instantiate a sentence. The case grammar formalism [Fil76] is used in order to define how verbs are associated with terms, which can be detailed by modifiers. Each term takes a certain thematic role around the verb; it can be an agent or theme of the sentence, for instance. Each case frame can also be associated to more than one verb, all of them assuming the same meaning (synonymous).

Figure 3.8 is the XML [RM01] structure used to define case frames.

```
<frame>
  <description></description>
  <name></name>
  <verb-list>
    <verb></verb>
    <verb></verb>
  </verb-list>
  <roles>
    <role mandatory=" ">agent</role>
    <role mandatory=" ">theme</role>
    <role mandatory=" ">from-value</role>
    <role mandatory=" ">to-value</role>
    <role mandatory=" ">from-loc</role>
    <role mandatory=" ">to-loc</role>
    <role mandatory=" ">at-loc</role>
    <role mandatory=" ">instrument</role>
  </roles>
</frame>
```

Figure 3.8 Case frame definition

Each *frame* tag contains a case frame definition. The *description* tag gives a brief explanation about how the case frames can se used, possibly including examples. The *name* tag identifies the case frame. The *verblist* tag contains a set of *verb* tags that refer to the verbs related to this case frame. The *verbs* from this list should have the same semantic meaning. Consequently, they own the same arguments defined in the *roles* tags. Each *role* tag determines a possible verb argument and its type. It has the *mandatory* attribute, which determines whether the argument is obligatory or not.

The following are the possible role types for *role* tag:

- agent: a term that executes the verb action.
- theme: a term that suffers the verb action.
- from-value: a term that determines the theme past value.
- to-value: a term that determines the theme future value.
- from-loc: a term that determines the theme past location.
- to-loc: a term that determines the theme future location.
- at-loc: a term that determines the theme current location.
- instrument: a term that determines the way the agent executes the verb action.

An example of a case frame definition is shown in Figure 3.9. The *Select Item* identifies the case frame defined by the *select* and the *choose* verbs. The *agent* and the *theme* are mandatory, thus they need to be specified when these verbs are used. The *from-loc* is optional. As a result, it is not necessary to specify this argument.

```
<frame>
  <description>Select an item from location.
  Example: Select the send message option from menu.
  </description>
  <name>SelectItem</name>
  <verb-list>
    <verb>select</verb>
    <verb>choose</verb>
  </verb-list>
  <roles>
    <role mandatory="true">agent</role>
    <role mandatory="true">theme</role>
    <role mandatory="false">from-loc</role>
  </roles>
</frame>
```

Figure 3.9 Case frame definition example

To illustrate how verbs are associated with their arguments, some examples of CNL sentences are shown in Figure 3.10.

Open the URL link with browser.	Open **<theme> <instrument>**
Exit from the message menu.	Exit **<from-loc>**
Return to saved messages folder.	Return **<to-loc>**
Phone **is** in message inbox screen.	**<theme>** is **<at-loc>**

Figure 3.10 CNL sentences examples

3.4. Case Frame Restriction

The case frame restriction defines the relation between verb's arguments and Ontology classes. Each verb's argument belongs to an Ontology class in order to restrict the way phrases are written. This minimizes the possibility of writing semantically wrong sentences.

Figure 3.11 shows the XML [RM01] structure used to define case frame restrictions.

```
<frame>
  <name></name>
  <restrictions>
    <restriction name=" ">
      <class role=" "> </class>
    </restriction>
    <restriction name=" ">
      <class role=" "> </class>
    </restriction>
  </restrictions>
</frame>
```

Figure 3.11 Case frame restriction definition

The *frame* tag defines a case frame restriction and its identification is captured by the *name* tag. It contains the *restrictions* tag that holds all possible restrictions. Each *restriction* tag contains an attribute *name* for identifies and a list of *class* tags. Each *class* tag defines the Ontology class associated with the verb's argument role.

The following Figures contain the case frame definition *SetItem* for the verbs *set* and *check* (Figure 3.12), and its respective case frame restriction (Figure 3.13).

```
<frame>
  <description>Set the value for an item.
               Ex.: Set the Fix Dialing to on
  </description>
  <name>SetItem</name>
  <verb-list>
    <verb>set</verb>
    <verb>check</verb>
  </verb-list>
  <roles>
    <role mandatory="true">agent</role>
    <role mandatory="true">theme</role>
    <role mandatory="false">to-value</role>
  </roles>
</frame>
```

Figure 3.12 Case frame definition SetItem

As can be noticed, the case frame *SetItem* contains the roles: *agent*, *theme* and *to-value.* So, for those three roles, there are four defined restrictions.

```
<frame>
  <name>SetItem</name>
  <restrictions>
    <restriction name="DTSET_FIELDVALUE_FIELD">
      <class role="theme">field</class>
      <class role="to-value">field_value</class>
    </restriction>
    <restriction name="DTSET_STATEVALUE_SENDABLEITEM">
      <class role="theme">sendable_item</class>
      <class role="to-value">state_value</class>
    </restriction>
    <restriction name="DTSET_STATEVALUE_ITEM">
      <class role="theme"> item</class>
      <class role="to-value">state_value</class>
    </restriction>
    <restriction  name="DTSET_ITEM">
      <class role="theme">item</class>
    </restriction>
  </restrictions>
</frame>
```

Figure 3.13 Case frame restriction definition SetItem

The first three roles restrict the *theme* and the *to-value* arguments, and the fourth restricts only the *theme* argument, once the *to-value* argument is not mandatory. Each restriction has a *name*, which is used to define a CSP *datatype*. To conclude the restriction definition, it is necessary to associate every role to an ontology class. This association restricts verb arguments, for instance the *DTSET_FIELDVALUE_FIELD* restriction defines that the *theme* is a term from the Ontology class *field* and the *to-value* argument belongs to the *field_value* class.

3.5. CSP Alphabet

In order to define use cases in natural language (CNL), it is necessary to specify CSP definitions, which are able to recognize and manipulate a CSP event. As mentioned in Section 2.1, CSP events are abstracts events that mean real actions, taken by the users or the system.

The result of mapping all structures, which are necessary for processing CSP structures to natural language, is a CSP file named *CSPHeader.* This file contains all definitions (*datatypes*, *tuples* and *channels*) used for interpreting CSP events as natural sentences in CNL.

In this section, is described how the *CSPHeader* file is generated.

3.5.1. Ontology class to CSP datatype

For each Ontology class, there is an associated specific CSP *datatype*. As can be seen in Figures 3.14 and 3.15, Ontology classes are mapped to CSP *datatypes*. The datatype elements refer to subclasses of a bigger class. It thus become clear that the same hierarchy level established on Ontology can be defined in the CSP *datatypes.*

```
<datatype class="value">
        <label/>
        <name>Value</name>
</datatype>

<datatype class="state_value">
        <label>STATEVALUE</label>
        <name>StateValue</name>
</datatype>

<datatype class="field_value">
        <label>FIELDVALUE</label>
        <name>FieldValue</name>
</datatype>
```

Figure 3.14 Ontology classes

```
datatype Value =
                DTSTATEVALUE.StateValue
              | DTFIELDVALUE.FieldValue

datatype StateValue =  state1 | state2 | ...

datatype FieldValue =  field1 | field2 | ...
```

Figure 3.15 CSP datatypes definition

The datatype *Value*, considering Figure 3.15, is a CSP *datatype* that can have a *StateValue*, or a *FieldValue.*

3.5.2. Case frame to channel

The thematic roles of case frames are represented as parameters of CSP *channels*. For each role and its set of restrictions, which are defined by case frame restriction, defined datatypes are used as parameters to the channels. Thus, the CSP channel that defines the case frame showed in Figure 3.12 can receive three parameters, which specify the thematic roles *agent, theme* and *to-value*. The CSP channels receive tuples in two arguments. First argument is the thematic role and the second one is a set of modifiers ({} in CSP).

The following figures show how a CSP *channel* is defined. The CSP channel *Set* was defined based on the case frame illustrated in Figure 3.12 and it can receive two kinds of parameters for one or two tuples either.

```
<channel>
  <name>set</name>
  <case-grammar>SetItem</case-grammar>
  <datatype>DTSet</datatype>
<channel>
```

Figure 3.16 Channel definition example in XML

```
channel Set : DTSet
datatype DTSet =
              DTSET_ITEM.(Item, Set(Modifier))
            | DTSET_ITEM.(Item, Set(Modifier)).(Item, Set(Modifier))
```

Figure 3.17 CSP channel definition

Using the definition in XML, the CSP *channel* is generated. The datatype *DTSet* is automatically created following the restrictions on case frame. Notice that the names of restriction are the same used on the channel definition.

4. CSP 2 CNL Tool

This chapter describes the automation strategy used to translate CSP models from both user and component view use cases into their corresponding use cases document, written in English CNL.

Section 4.1 discusses some architecture issues related to the tool design. In Section 4.2, we introduce an overview of the tool which translates CSP into CNL.

4.1. Architecture

The CSP 2 CNL tool was implemented using Java language [Gra97] and Eclipse platform [CR04]. The purpose of using Java is to guarantee executions in multiple platforms. The Eclipse platform is an open and freeware platform that is useful to give support to build IDEs (Integrated Development Environments) that can be used to create a great variety of applications, such as web-sites, Java, C or C++ programs, applications to Embedded Systems, and many others.

The Java language and the Eclipse platform were chosen because they were the programming language and the development platform used by the tool implemented by [Cab06].

In order to better visualize the class diagram, it was divided into two class diagrams: one representing the relation between the classes involved in the user view use cases generation, and another diagram to represent the relation between the classes involved in the component view use cases generation. Component and user view uses cases are explained in Section 4.2. Figures 4.1 and 4.2 shows the class diagrams created.

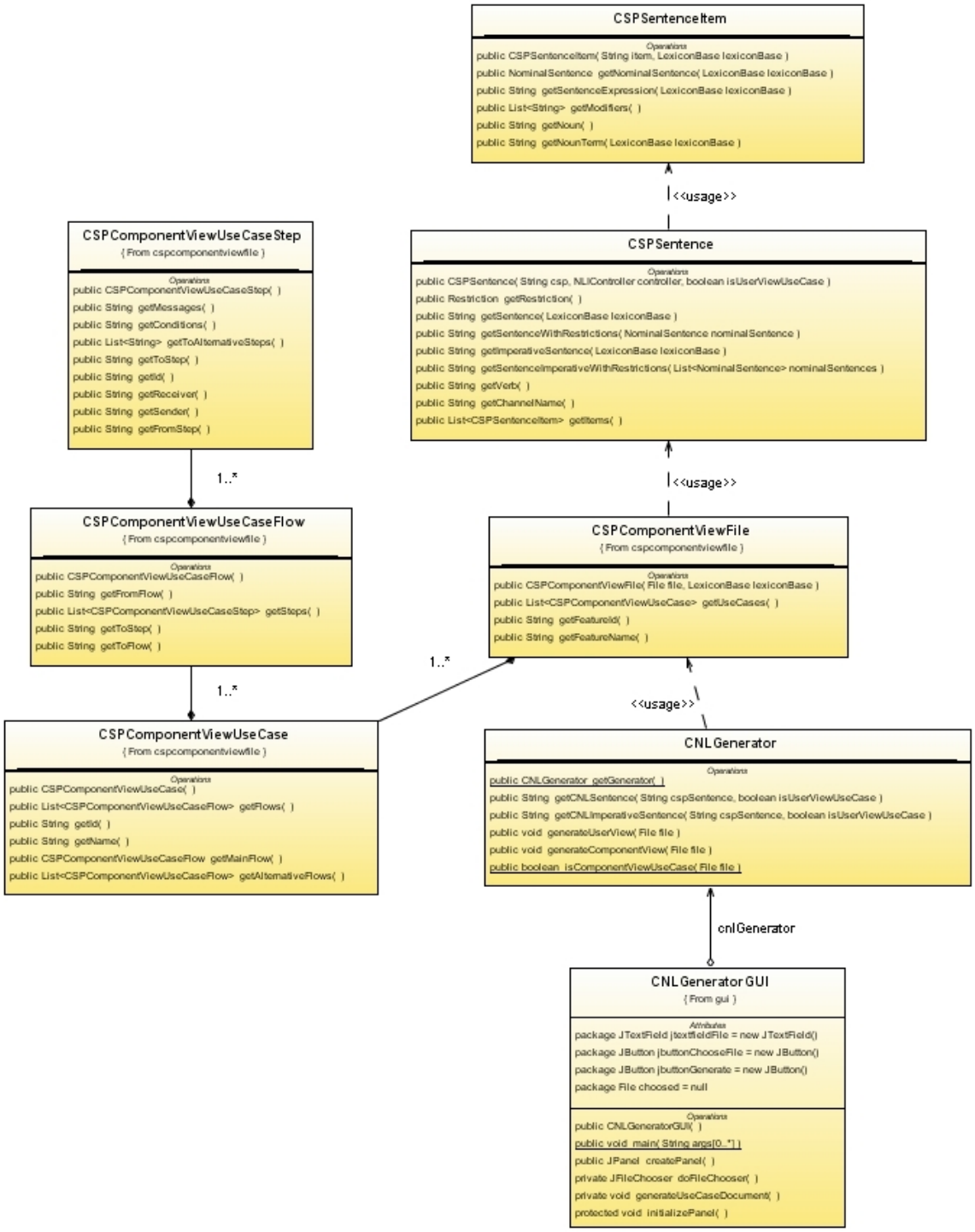

Figure 4.1 Component view generation class diagram

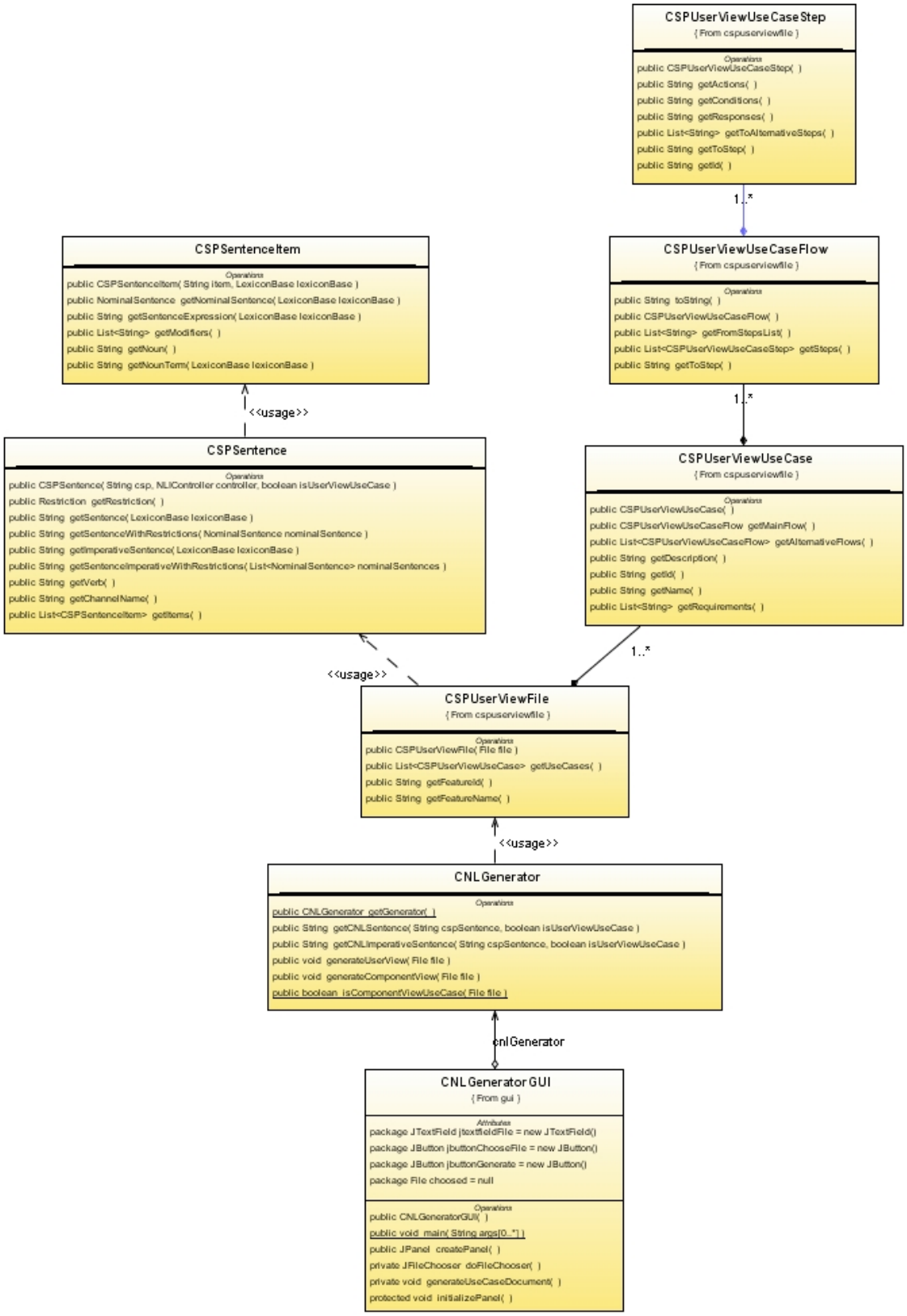

Figure 4.2 User view generation class diagram

The *CNLGeneratorGUI* class is where the implementation of our tool graphic's interface takes place. It has an instance of the *CNLGenerator* class which processes the translation from CSP to CNL and generates the Word 2003 documents files. In order to effectively translate component and user view files the *CNLGenerator* verifies if the input file is an instance of a *CSPUserViewFile* or of a *CSPComponentViewFile*, and delegates the transformation for one of them.

Once the input file's instance is checked, its related class (*CSPComponentViewFile* or *CSPUserViewFile*) starts the transformation. In parallel, it does the file's parsing while creating the use cases (*CSPComponentViewUseCase or CSPUserViewUseCase*), flows (*CSPComponentView-UseCaseFlow or CSPUserViewUseCaseFlow*) and steps (*CSPComponentViewUseCaseStep or CSPUserViewUseCase*Step) objects. During the file's parsing, the CSP sentences are kept in *CSPSentence* objects, where they are translated to CNL sentences and have the right restrictions applied to their verb's arguments and modifiers, which are represented by a list of *CSPSentenceItem*.

As the use cases, flows and steps objects are filled with the information that came from the input CSP file, and the steps sentences are already translated to CNL, they are ready to be written in Word 2003 documents. The *CNLGenerator* receives these objects and apply their attributes to the corresponding (component view or user view) template document, which are written using Velocity, as it is explained in Section 4.2.3.

4.2. Overview

The CSP 2 CNL tool, illustrated in Figure 4.3 translates both user and component views use cases documents written in the CSP notation into their corresponding Microsoft Word 2003 [LLM04] documents, written in English.

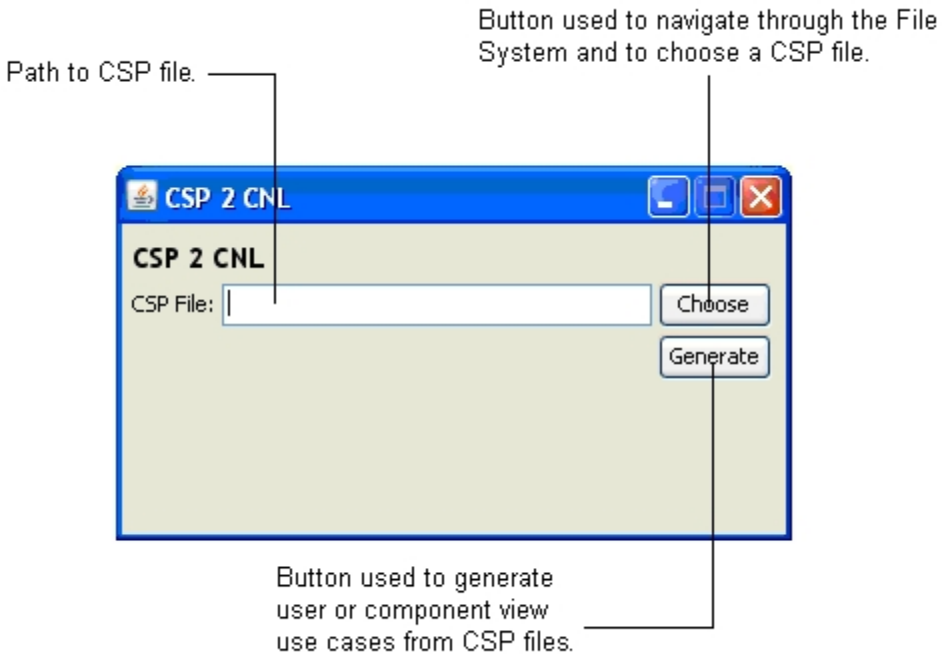

Figure 4.3 CSP 2 CNL tool main screen

The main difference between a user view use case and a component view use case is that the first one is generated from requirements documents and the second one is generated from architecture documents. The user view use cases clearly describe the system behavior when one single user executes it, by specifying the user operations and expected system responses. In the other hand, a component view use case specifies the system behavior based on the user interaction with the system components. In this view, the system is decomposed into components that concurrently process the user requests and communicate among themselves.

For each user view use case, it can be defined a related component view use case and user view steps are decomposed into component messages exchange. In the component view, it is defined the component that is invoking an action and the one that is providing the service. It is a message exchange process composed by a sender, a receiver and a message. The user from the user view use case is viewed here as a component, and can either send or receive messages to and from components, respectively. A component can also send a message to itself. These particularities enable the definition of concurrent scenarios, which is a non-functional requirement. Thus, components can share resources and exchange messages.

Both user and component views use cases are translated to their formal model specifications in CSP by the Use Model Generator tool proposed in [Cab06]. The use cases are first written according to CNL grammar, which is defined by knowledge bases, before the translation to

their corresponding CSP models. The method that makes it possible to translate each CNL sentence from the use cases templates into CSP events is defined in [Cab06].

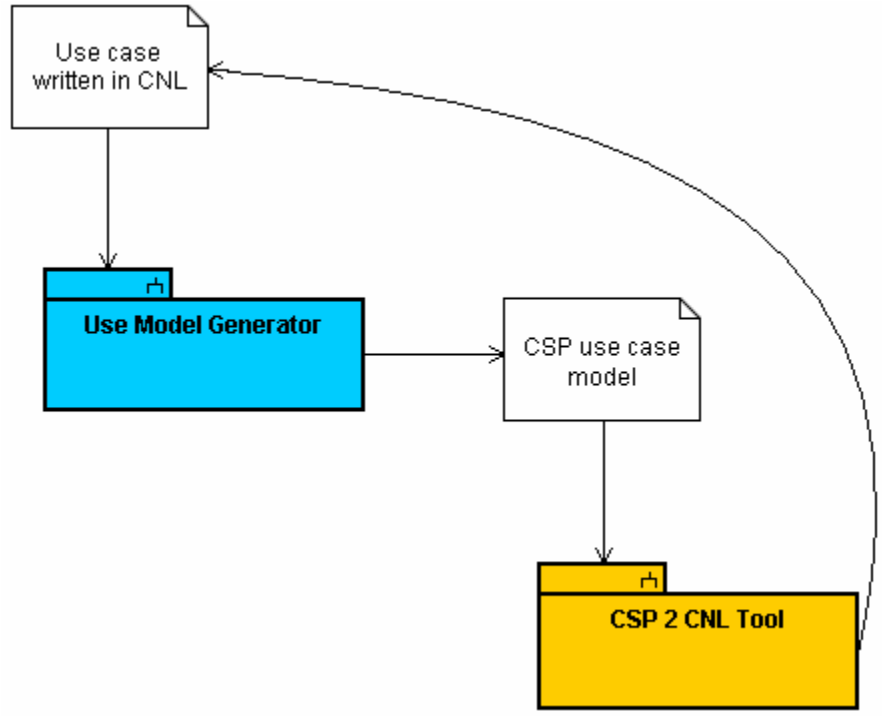

Figure 4.4 CSP 2 CNL tool scenario

Once the CSP models are created, they become the input to the CSP 2 CNL tool, as can be seen in Figure 4.4. Due to the fact that either the user view or the component view use cases represents different views of the system requirements, they also have distinct CNL template documents representations. The Use Model Generator tool receives each template document and generates different CSP model structures descriptions with particularities that have to be treated separately in order to generate their corresponding CNL constructions.

The CSP 2 CNL tool gives a different treatment to both user view and component view CSP models generated by the Use Model Generator in order to transform them into their correspondents CNL use case Word 2003 documents templates.

The following subsections summarize the functionalities the CSP 2 CNL tool performs.

4.1.1. User view use case generation

In order to generate the user view use case document, written in CNL, it is necessary that the user of the CSP 2 CNL tool chooses an user view CSP file, as it is illustrated in Figure 4.5.

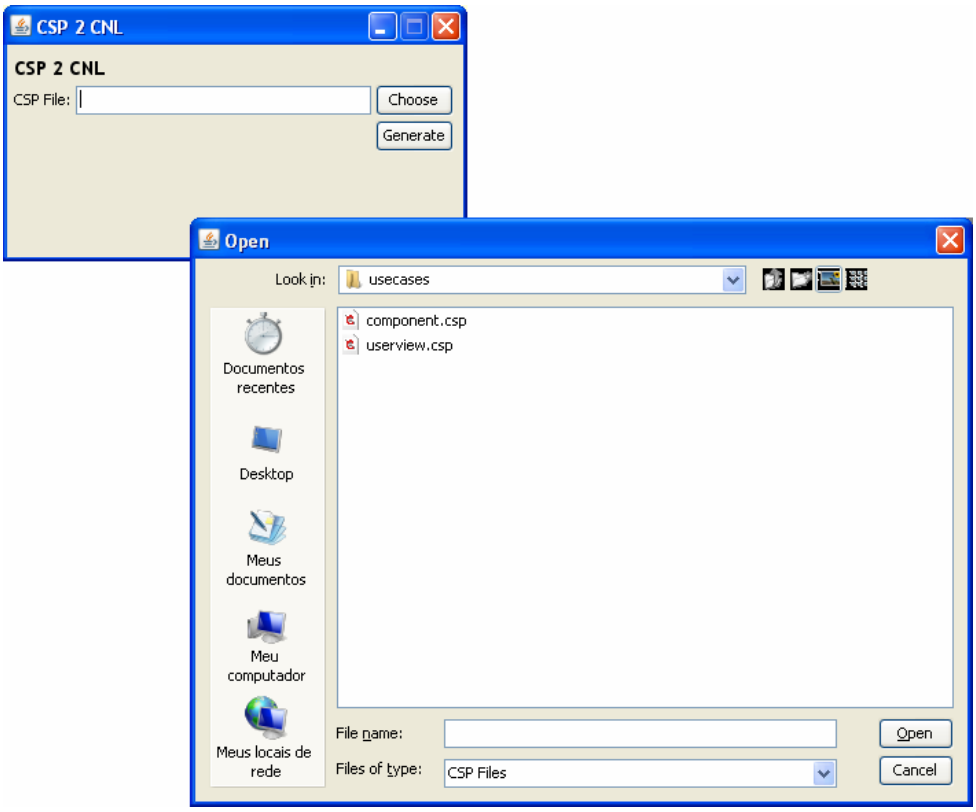

Figure 4.5 Choosing a csp file

After the file selection and the Generate button is pressed, the tool automatically starts to execute the transformation. Figure 4.6 shows an example of a user view CSP file.

```
include "CSP_HEADER_USER.csp"

channel steps, conditions, expectedResults

System = UC_01_1M ; System

-- Feature: 11166 - IM conversation Structured Data

-- Use Case: IM Main Functionalities
-- Description:
        This use case starts a conversation with a contact in the contact list.

-- Requirements: TRS 14293
UC_01_1M =

        -- Start IM application.
        steps ->
        start.DTSTA_APPLICATION.(IM_APPLICATION, {}) ->

        -- IM application is displayed.
        expectedResults ->
        isstate.DTISS_APPLICATION_STATEVALUE.(IM_APPLICATION, {}).(DISPLAYED_VALUE, {}) ->

        ( UC_01_2M [] UC_01_1A )

UC_01_2M =

        -- Log in IM Server.
        steps ->
        login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION, {}) ->

        -- User is logged in.
        expectedResults ->
        isstate.DTISS_USER_STATEVALUE.(USER, {}).(LOGGED_IN_STATE, {}) ->

        UC_01_3M

UC_01_3M =

        -- Select all contacts. Start conversation.
        steps ->
        select.DTSEL_LISTITEM.(CONTACT_ITEM, {ALL})
              -> start.DTSTA_LISTITEM.(CONVERSATION_ITEM, {}) ->

        -- Conversation screen is displayed. IM editor is empty.
        expectedResults ->
        isstate.DTISS_SCREEN_STATEVALUE.(CONVERSATION_SCREEN, {}).(DISPLAYED_VALUE, {})
              -> isstate.DTISS_FIELD_STATEVALUE.(IM_EDITOR_FIELD, {}).(EMPTY_STATE, {}) ->

        SKIP
```

Figure 4.6 User view CSP model example

The tool reads the file in order to identify whether it's a user or component view representation. It is done by checking via a regular expression if the term `Comp` is presented inside the file. A more detailed explanation about the use of the term `Comp` is presented in Section 4.2.2. If the term is not found, then the current CSP file is an instance of a user view file.

Next, the tool executes the file parser, keeping in Java objects the information about its use case(s). A major object is created to keep the feature information and the use case(s). Each use case is treated as an object that keeps the use case id, name, and flows. The use case flows are also represented as an object that stores the information regarding to the flows in order to make it

possible to organize all the steps, found in the CSP file, into main and alternatives flows. Each flow has its From Step and To Step attributes, and its set of sequential steps.

Then, for each step of the flows, three possible CNL sentences types are generated: one for the **User Action**, another, if any, representing the **System State**, and another for the **System Response**. Using the action sentence `Select all contacts` in step `UC_01_3M` as an example of sentence in Figure 4.6, the CNL generation is done as follows:

- The sentence's verb (`Select`) is found by the channel `select`.
- The restriction `DTSEL_LISTITEM` is applied to the verb's argument `(CONTACT_ITEM, {ALL})`. The verb's argument is also translated to CNL. In this case, `CONTACT_ITEM` becomes `contact` in CNL. The modifier `ALL` is also translated to the CNL term `all`. The xml specification of the modifier in the CNL Lexicon base defines how the modifier will be appended to the sentence. In this case, the definition of the modifier `ALL` says that it comes before the term it modifies and makes the term go to its plural form. At this point, the CSP sentence is already translated to its corresponding CNL sentence: `Select all contacts.`

As the use cases, flows and steps were already organized in objects, which represents the links between feature and use cases, use case and flows, and flows and steps, finally the tool is apt to structure the Word 2003 document, summarized in Section 4.2.3. The next figure, illustrates the use case main flow generated by the component csp file showed in Figure 4.6.

Feature 11166

Use Cases

UC_01 - IM Main Functionalities

Related requirement(s)

Requirements Codes
TRS 14293

Description

This use case starts a conversation with a contact in the contact list.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Start IM Application.		IM Application is displayed.
UC_01_2M	Log in IM Server.		User is logged in.
UC_01_3M	Select all contacts. Start conversation.		Conversation screen is displayed. IM editor is empty.

Alternative Flows

From Step: UC_01_1M
To Step: SKIP

Step Id	User Action	System State	System Response
UC_01_1A	Go to Saved Conversations folder.		Saved Conversations folder is displayed.

Figure 4.7 User view use case generated

When the transformation from CSP to CNL terminates, the CSP 2 CNL user is informed as can be seen in Figure 4.8. The user view use case document is now written in the English CNL.

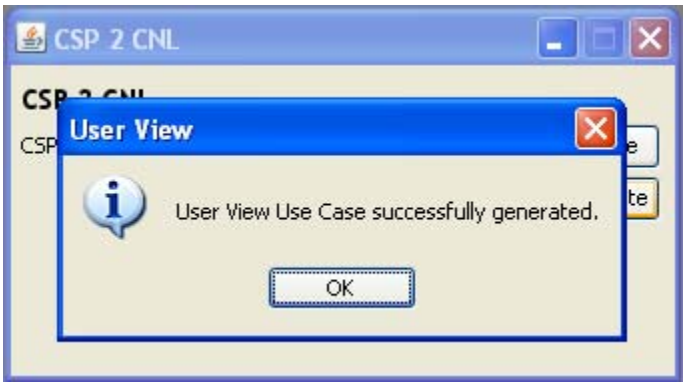

Figure 4.8 End of user view use case generation

4.1.2. Component view use case generation

As it was mentioned before, the treatment given to component view CSP models, so that it is possible to translate them into use cases CNL documents, is quite different from the one given to user view CSP models. It happens because the component CSP model structure itself is different from the one adopted in the user CSP. This difference is well described in Chapter 4 of [Cab06].

Figure 4.9 shows an example of a component CSP. One of the differences that can be noticed is that the channels now have their names suffixed by `Comp`, making the user and component view CSP alphabets different.

Each component process is defined as a sequence of messages communicated with other component. Figure 4.9 is illustrating only the `USER_P` process, which groups all the messages exchanged into the `USER` and the other components for the presented use case. The remaining processes, that groups the messages related to other use case components, is structured in the same way that the `USER_P`.

For every component use case steps there are now two CSP events. Each CSP event shows the sender and receiver components involved in its message exchange. In Figure 4.9, the message `Read incoming message` has the CSP notation in the `USER_UC_02` process as `readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, { })`. Between the channel name (`readComp`) and the restriction name (`DTREA_SENDABLEITEM`) comes the sender (`USER`) and receiver (`MESSAGE_APP`) components.

```
Scenarios = (USER_P); Scenarios

include "CSP_HEADER_COMPONENT.csp"

System = USER_P ||| MESSAGE_APP_P ||| MESSAGE_VIEWER_P ||| MENU_CONTROLLER_P
          ||| MESSAGE_STORAGE_APP_P ||| LIST_APP_P ||| DISPLAY_APP_P

-- Feature: 12898

-- UseCase: UC_02 - Incoming message is moved to the Important Messages folder

USER_P =
  -- Scenario Case: Incoming message is moved to the Important Messages folder.
  USER_UC_02

USER_UC_02 =
  -- Step: UC_02_1M - Message: Read incoming message.
  readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{}) ->
  -- Step: UC_02_3M - Message: Open menu.
  openComp.USER.MESSAGE_APP.DTOPE_MENU.(MENU, {}) ->
  -- Step: UC_02_5M - Message:  Move to Important Messages option is displayed.
  isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.(MOVE_TO_IMPORTANT_MESSAGES_OPTION, {})
   .(DISPLAYED_VALUE, {}) ->
  -- Step: UC_02_6M - Message: Select Move to Important Messages option.
  selectComp.USER.MESSAGE_APP.DTSEL_MENUITEM.(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}) ->
  (USER_UC_02_9M [] USER_UC_02_3E)

USER_UC_02_9M =
  -- Step: UC_02_9M - Message:  Message moved to Important Message folder is displayed.
  isstateComp.MESSAGE_STORAGE_APP.USER.DTISS_DIALOG_STATEVALUE
   .(MESSAGE_MOVED_TO_IMPORTANT_MESSAGES_FOLDER, {}).(DISPLAYED_VALUE, {}) ->
  -- Step: UC_02_10M - Message: Wait for at most 2 seconds.
  waitComp.USER.USER.DTWAI_ITEM.(SECOND, {AT_MOST.2}) ->
  -- Step: UC_02_12M - Message: Available message is selected.
  isstateComp.LIST_APP.USER.DTISS_SENDABLEITEM_STATEVALUE.(AVAILABLE_MESSAGE, {})
    .(SELECTED_VALUE, {}) ->
   USER_P

USER_UC_02_3E =
  -- Step: UC_02_3E - Message: Confirm memory information dialog.
  confirmComp.USER.MESSAGE_APP.DTCON_DIALOG.(MEMORY_INFORMATION_DIALOG,{}) ->
  -- Step: UC_02_4E - Message: Message content is displayed.
  isstateComp.MESSAGE_APP.USER.DTISS_FIELDVALUE_STATEVALUE.(MESSAGE_CONTENT_FIELD_VALUE, {})
    .(DISPLAYED_VALUE, {}) ->
   USER_P
```

Figure 4.9 Component view CSP model example

In order to transform the component CSP into its corresponding Word 2003 document template, the component CSP file needs to be selected in the CSP 2 CNL main screen in the same way as the user view csp file, as it was showed in Figure 4.5. After the file selection and the Generate button is pressed, the tool automatically starts the transformation.

The tool reads the file in order to identify whether it's a user or component view representation. It is done by simply looking for the term `Comp` using a regular expression. The expression differentiates the term `Comp`, which comes appended to CSP channel names, from other similar terms. If the term is found, then the current CSP file is an instance of a component view file.

Next, the tool executes the file parser, keeping in Java objects the information about its use case(s). A major object is created to keep the feature information and the use case(s). Each use

case is an object that keeps the use case id, name, and flows. The use case flows are also represented as an object that stores the information regarding to the flows in order to make it possible to organize all the steps, found in the CSP file, into main and exceptions flows. Each flow has its From Flow and To Flow attributes, and its set of sequential steps.

Then, for each step of the flows, two possible CNL sentences are generated: one for the **Message** exchanged by the components, and another, if any, for the representation of the **System State**. Using the message `Read incoming message` in Figure 4.9 as an example, the CNL generation is done as follows:

- The sentence's verb (`Read`) is found by the channel `readComp`.
- The sender (`User`) and receiver (`Message App`) components are extracted from the CSP sentence by the terms `USER` and `MESSAGE_APP`.
- The restriction `DTREA_SENDABLEITEM` is applied to the verb's argument (`INCOMING_MESSAGE, { }`). The verb's argument is also translated to CNL. In this case, `INCOMING_MESSAGE` becomes `incoming message` in CNL. If there were modifiers to the verb's arguments, they would also be translated to CNL and appended to the sentence regarding their definition in the CNL Lexicon base. At this point, the CSP sentence is already translated to its corresponding CNL sentence: `Read incoming message`.

As the use cases, flows and steps were already organized in objects, which represents the links between feature and use cases, use case and flows, and flows and steps, finally the tool is apt to structure the Word 2003 document, summarized in Section 4.2.3. Figure 4.10 illustrates the use case main flow generated by the component csp file showed in Figure 4.9.

Feature 12898

UC 02 - Incoming message is moved to the Important Messages folder

Main Flow

From Flow: START
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1M	User	Read incoming message.		Message App
UC_02_2M	Message App	Open incoming message.		Message Viewer
UC_02_3M	User	Open menu.		Message App
UC_02_4M	Message App	Display menu.	Important Messages feature is on.	Menu Controller
UC_02_5M	Menu Controller	Move to Important Messages option is displayed.		User
UC_02_6M	User	Select Move to Important Messages option.		Message App
UC_02_7M	User	Select Move to Important Messages option.		Message App
UC_02_8M	Menu Controller	Save message to Important Messsages folder.	Message storage is not full.	Message Storage App
UC_02_9M	Message Storage App	Message moved to Important Message folder is displayed.		User
UC_02_10M	User	Wait for at most 2 seconds.		User
UC_02_11M	Message App	Next inbox message is highlighted.		List App
UC_02_12M	List App	Avaiable message is selected.		User

Figure 4.10 Component view use case main flow

When the transformation from CSP to CNL terminates, the CSP 2 CNL user is informed as can be seen in Figure 4.11. The component view use case document is now written in the English CNL.

Figure 4.11 End of component view use case generation

4.1.3. Word 2003 document generation

In order to write user and component view use cases into Word documents, the tool utilizes the both use cases templates defined in Chapter 3 of [Cab06].

Our generation strategy was to get the XML Word representation for both templates and work with them. This representation is called WordprocessingML [MC03], also known as WordML, which is the XML file format for Microsoft Word 2003 documents. These documents are used to create templates with Velocity [Vel07], a tool to generate text that aggregates Java object information with the files described in WordML. Then, Velocity specific demarcations are added to the WordML files. These demarcations will be properly replaced with their related Java objects information.

With the use cases, flows and steps organized in Java objects, as it was explained in Sections 4.2.1 and 4.2.2, it was easy to compose the Word document.

4.3. Some considerations

In order to make it possible the presented tool successfully translates CSP models use cases files into Word 2003 use cases documents, some changes were made in the CSP files generated by [Cab06], which, in fact, have become our input files to the examples posted here and to the experiments in Chapter 5. These modifications were taken as follows:

User view CSP files:

- The only field added to the user view CSP files was `-- Feature: XXXX`, where XXXX means the feature id. This field was added before the use cases specifications in order to retrieve the feature ids.

Component view CSP files:

- In order to retrieve the feature ids, the field `-- Feature: XXXX`, where `XXXX` actually means the feature id, was also added before the use cases specifications.
- Line `-- UC_XX - useCaseName` is introduced, where `UC_XX` is the use case id and `useCaseName` is the place where the use case name appears. This line is the first line of every new use case of every CSP file. This allows us to find the use cases id and name.
- Before each step sentence written in CNL, the field `-- Step: UC_XX_YY` was added, so that the step id, related to the step where the sentence came from in the original use case document, could be retrieved.
- The tag `_Start` was added to every reference of an exception flow's first step in order to make it possible to find the beginning of each flow. Whenever a step is some flow's first step, its reference in the CSP file appears as `UC_XX_YY_Start`.

The previous modifications were necessary to avoid losing the original use cases specifications information after their transformation into CSP models by the tool developed in [Cab06]. Without considering the previous modifications, it would be impossible to create use cases documents with the same information presented in the original documents.

5. Case Study

This chapter summarizes the results of some experiments that were made in the Motorola's context. These experiments aim to evaluate the effectiveness of the automation strategy achieved by the tool presented in the previous chapter. Their mainly goal is to show examples of usage of the tool related to its input and output points of view. Hence, two experiments have been accomplished, involving real Motorola's requirements documents.

Section 5.1 explains how the experiments have been executed and Section 5.2 lists the main points of their results in a brief and comprehensive manner.

5.1. Experiments

The experiments made included requirements documents that contain both user and component view uses cases. For each experiment it was defined different use cases. However, all use cases were essentially the same size, including a main execution flow and at least one alternative or exception flow.

For the first experiment, two requirements documents were chosen. Each one contained only one use case from a different view of the system.

The second experiment was composed of two different requirements documents, including two use cases each.

5.1.1. First experiment

For the first experiment execution, there were chosen two simple requirements documents, one containing a user view use case, and another containing a component view use case. There were chosen simple documents structures so that it could be possible to verify if our proposed tool is, in fact, translating the use cases steps sentences from CSP to CNL and organizing the documents as they are organized in the templates defined by [Cab06]. The original Motorola's documents selected, written in these templates, can be seen in Appendix A, Sections A.1 and A.3.

At the beginning of the experiment, the Use Model Generator tool developed by [Cab06] was used in order to generate the CSP files, containing the use cases formal models. Fragments of these files are illustrated in Chapter 4, Figures 4.6 and 4.9.

Once the use cases formal models were created, they became our CSP 2 CNL inputs. After executing our tool, it successfully translated the CSP files into Word 2003 documents files written in English CNL, as can be seen, in their complete version, in Appendix A, Sections A.2 and A.3.

5.1.2. Second experiment

For the second experiment execution, there were chosen two requirements documents, one containing two user view use cases, and another containing two component view use cases.

There were chosen more elaborated requirements documents with two use cases each in order to verify if our proposed tool is not only translating the use cases steps sentences from CSP to CNL and organizing the documents following the defined templates, but also to verify that it is also treating documents with more than one use case and organizing them, keeping their steps and flows links as they appear in the original Motorola's documents. These documents can be seen in Appendix B, Section B.1 and B.3.

At the beginning of the experiment, in the same way as the first experiment did, the Use Model Generator tool developed by [Cab06] was used in order to generate the CSP files, containing the use cases formal models. Figures 4.6 and 4.9, in Chapter 4, show fragments of the CSP files used.

Once the use cases formal models were created, again they became our CSP 2 CNL inputs. After executing our tool, it has successfully translated the CSP files into Word 2003 documents files written in English CNL, as can be seen in Appendix B, Sections B.2 and B.3.

5.2. Obtained results

The two experiments have had satisfactory results as can be verified by the comparison between Motorola's documents and the ones generated by CSP 2 CNL tool, that are showed in Appendix at the end of this work.

Although the documents generated by our tool have some mismatched format structures, for instance different font types in some words, and some different arranging of the spaces between flows, in general they are very similar to the templates used by Motorola's documents.

The main automation strategy was achieved as the documents generated by our tool show the uses cases, with the right connections between their flows and steps, in accordance to the original documents.

6. Conclusion

When comparing the development of software systems to other projects types, as engineering projects for instance, it is a fact that they present different challenges to be achieved. Software systems may suffer frequent changes in their specifications throughout their life cycle as their development projects are prone to change. Original requirements almost always change after activities of design or implementation have started. No matter what the reasons are, whether external, internal, technical or learning, requirements change. That is a factor of life. Besides, software projects have the capacity to allow changes that engineering projects do not have. Renaming an artifact, throwing away a project prototype, or even deleting some parts of what is not important anymore for the project are characteristics not presented in the physic reality of Civil Engineering projects, for instance. Throwing away prototypes is complicated, even to elaborate them is a tough work to be accomplished by engineers. It is true that they build houses for a longer time than we develop software, so it is reasonable that Engineering has stronger established concepts than we do.

Even though the idea that something has to change, most of time, brings apprehension, changing requirements need to stop to be seen as a threat and start to be seen as an opportunity. If the requirements of a software system do not change, this means that they are fixed and known. The truth is that anyone can implement them, even their competitors. In other words, the requirements stop being “utility” to become “commodity”, they are not anymore what makes a difference between a software and another. Keeping well informed with the evolution of requirements is critical as requirements represent the rationale behind development and the milestones against which project success is measured.

As requirements are the agreement between clients and developers, writing their documents in a controlled natural language has demonstrated that their resulting artifacts are non-ambiguous and better understood by the ones involved with the software system development. Besides, the requirements changes management becomes less painful, as they are clearly written.

However, dealing with changes in requirements means that a set of related documents has also to be changed. And that is a pain in the neck. It takes time, and usually, it wastes more time than expected.

As software developers, our work is to create mechanisms that make it possible to automate tasks to our clients. It is more than natural that we bring to our environment automations that help us to develop software with more quality and speed. Working with the requirements changes in an automated way it is nothing else than to use the philosophy that we use with our clients. It seems to be a great catch to automate the requirements maintenance so that their

documents would be frequently updated as soon as new changes arise. The use of formal models has become a hand on the wheel to specify requirements and to enable automation.

Our work presented an approach to automate the translation form formal models specifications, written in CSP notation, to requirements documents, written in an English controlled natural language. The CSP 2 CNL tool implemented makes it possible to keep in track with requirements changes as they happen during the entire software life cycle. It has demonstrated to be effective dealing with CSP models files so that they could be transformed into use cases Word 2003 documents written in English.

6.1. Future works

This section contains some suggested improvements to make the use of our proposed approach more practical.

6.1.1. CSP models standardization

The automation strategy presented in this work and implemented by the tool CSP 2 CNL is the link that was missing between the tools developed by [Cab06] and [Sou06] so they could work together. These last two tools constitutes a part of a major project inserted in Motorola's context, where there were proposed some strategies to automate the process of keeping requirements documents updated by their related test cases.

Thus, in order to make these three tools work together, it is necessary that the CSP models files generated by [Cab06] and the ones generated by [Sou06] are created following the same format. Currently, as these projects were developed separately, the CSP models created by them are structured in different formats.

As the input CSP models files of our tool use the format adopted and created by [Cab06], a suggested future work is to get the CSP files generated by [Sou06] and modify their format so that it become the same format as our tool input files.

6.1.2. Automatic requirements update by test cases

Another suggestion for a future work is to make it possible that our CSP 2 CNL tool works attached to the tools developed by [Cab06] and [Sou06], so that the transformation from CSP to CNL could be executed automatically, without users interaction.

As soon as a CSP model is generated by the tool in [Sou06], which translates test cases into their corresponding CSP models, the CSP 2 CNL tool should transform it to its corresponding

use case written in English CNL. Thus, the use cases documents could be always updated at the moment new changes are assigned to their corresponding test cases.

6.1.3. Modify CSP file format

As the Use Model Generator tool developed by [Cab06] generates CSP files, which formats were adopted as our tool input files formats, and some important information about the use cases they belong to is lost after the transformation to CSP models, it is necessary the addition of some fields into these files. This necessity is explained in Chapter 4, Section 4.3.

Thus, another suggestion for a future work is to modify the CSP files generated by [Cab06] so that they present the missing information.

Bibliography

[Cab06] G. F. L. Cabral, (2006) *Geração de Especificação Formal de Sistemas a partir de Documento de Requisitos*. Master's Thesis, Universidade Federal de Pernambuco (UFPE).

[CIUFPE07] Centro de Informática da Universidade Federal de Pernambuco (CIn/UFPE). http://www.cin.ufpe.br, 2007.

[CR04] E. Clayberg and D. Rubel, (2004) *Eclipse: Building Commercial-Quality Plug-Ins*.

[Fil76] C.J. Fillmore, (1976) *Frame semantics and the nature of language*. In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech.

[Gra97] M. Grand, (1997) *Java Language Reference*. O'Reilly, 2nd edition.

[Hoa85] C. A. R. Hoare, (2004) *Communicating Sequential Processes*.

[HT99] A. Hunt and D. Thomas, (1999) *The Pragmatic Programmer: From Journeyman to Master*. Addison Wesley.

[Inc07] Motorola Inc http://www.motorola.com.br, 2007.

[Lei06] D. A. Leitão, (2006) *Nlforspec: Uma ferramenta para geração de especificações formais a partir de casos de teste em linguagem natural*. Master's thesis, Universidade Federal de Pernambuco (UFPE).

[LLM04] S. S. Laurent, E. Lenz and M. McRae, (2004) *Office 2003 XML: Integrating Office with the rest of the world.* O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[MC03] Microsoft Corporation, (2003) *Overview of WordprocessingML.*

[RM01] E. T. Ray and C. R. Maden, (2001) *Learning XML*.

[Ros97] A. W. Roscoe, (1997) *The Theory and Practice of Concurrency*.

[Sch99] S. Schneider, (1999) *Concurrent and Real-time Systems: The CSP Approach.*

[SLH03] R. Schwitter, A. Ljungberg, and D. Hood, (2003) *ECOLE - a look-ahead editor for a controlled language, in: Controlled translation.* In Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop, Proceedings of EAMT-CLAW03, May 15-17, Dublin City University, Ireland.

[Som03] I. Sommerville, (2003) *Engenharia de Software.* Addison Wesley, 6th edition.

[Sou06] C. F. Sousa, (2006) *Modelling and Integrating Formal models: from Test Cases and Requirements Models.* Master's Thesis, Universidade Federal de Pernambuco (UFPE).

[Vel07] The Apache Velocity Project http://velocity.apache.org/, 2007.

Appendix

Here, we present the documents involved in the experiments made in the case study presented in Chapter 5. The documents are exposed in their integral format, without none modifications, keeping the arranging of spaces as they are in the original documents.

For better visualization, the original Motorola's documents are emphasized in blue, and the documents generated by our tool are emphasized in green.

Appendix A - First experiment's documents

A.1. Motorola's user view document

Feature 11166 - IM conversation Structured Data

Use Cases

UC 01 - IM Main Functionalities

Related requirement(s)

Requirements Codes
TRS 14293

Description

This use case starts a conversation with a contact in the contact list.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Start IM application.		IM application is displayed.
UC_01_2M	Log in IM Server.		User is logged in.
UC_01_3M	Select all contacts. Start conversation.		Conversation screen is displayed. IM editor is empty.

Alternative Flows

From Step: UC_01_1M
To Step: END

Step Id	User Action	System State	System Response
UC_01_1A	Go to Saved Conversations folder.		Saved Conversations folder is displayed.

A.2. User view document generated by CSP 2 CNL tool

Feature 11166

Use Cases

UC_01 - IM Main Functionalities

Related requirement(s)

Requirements Codes
TRS_14293

Description

This use case starts a conversation with a contact in the contact list.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Start IM application.		IM application is displayed.
UC_01_2M	Log in IM Server.		User is logged in.
UC_01_3M	Select all contacts. Start conversation.		Conversation screen is displayed. IM editor is empty.

Alternative Flows

From Step: UC_01_1M
To Step: END

Step Id	User Action	System State	System Response

A.3. Motorola's component view document

Feature - 12898

Use Cases

UC 02 - Incoming message is moved to the Important Messages folder

Main Flow

From Flow: START
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1M	User	Read incoming message.		Message App
UC_02_2M	Message App	Open incoming message.		Message Viewer
UC_02_3M	User	Open menu.		Message App
UC_02_4M	Message App	Display menu.		Menu Controller
UC_02_5M	Menu Controller	Move to Important Messages option is displayed.		User
UC_02_6M	User	Select the Move to Important Messages option.		Message App
UC_02_7M	Message App	Select the Move to Important Messages option.		Menu Controller
UC_02_8M	Menu Controller	Save message to Important Messages folder.	Message storage is not full	Message Storage App
UC_02_9M	Message Storage App	Message moved to Important Message folder is displayed.		User
UC_02_10M	User	Wait for at most 2 seconds.		User
UC_02_11M	Message App	Next inbox message is highlighted.		List App
UC_02_12M	List App	Available message is selected.		User

Exception Flows

From Flow: UC_02_7M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1E	Menu Controller	Save message to Important Messages folder	Message Storage is full	Message Storage App
UC_02_2E	Message Storage App	Memory required message is displayed		Display App
UC_02_3E	User	Confirm memory information dialog		Message App
UC_02_4E	Message App	Message is displayed		User

A.4. Component view document generated by CSP 2 CNL tool

Feature – 12898

UC_02 - Incoming message is moved to the Important Messages folder

Main Flow

From Flow: START
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1M	User	Read incoming message.		Message App
UC_02_2M	Message App	Open incoming message.		Message Viewer
UC_02_3M	User	Open csm menu list.		Message App
UC_02_4M	Message App	Display csm menu list.	Important Messages feature is on.	Menu Controller
UC_02_5M	Menu Controller	Move to Important Messages option is displayed.		User
UC_02_6M	User	Select Move to Important Messages option.		Message App
UC_02_7M	User	Select Move to Important Messages option.		Message App
UC_02_8M	Menu Controller	Save message to Important Messsages folder.	Message storage is not full.	Message Storage App
UC_02_9M	Message Storage App	Message moved to Important Message folder is displayed.		User
UC_02_10M	User	Wait for at most 2 seconds.		User
UC_02_11M	Message App	Next inbox message is highlighted.		List App
UC_02_12M	List App	Avaiable message is selected.		User

Exception Flows

From Flow: UC_02_7M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1E	Menu	Save message to	Message storage is	Message

	Controller	Important Messsages folder.	full.	Storage App
UC_02_2E	Message Storage App	Memory required message is displayed.		Display App
UC_02_3E	User	Confirm memory information dialog.		Message App
UC_02_4E	Message App	Message content is displayed.		User

Appendix B - Second experiment's documents

B.1. Motorola's user view document

Feature 11166 - IM conversation Structured Data

Use Cases

UC 01 - IM Main Functionalities

Related requirement(s)

Requirements Codes
TRS 14293

Description

This use case starts a conversation with a contact in the contact list.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Start IM application.		IM application is displayed.
UC_01_2M	Log in IM Server.		User is logged in.
UC_01_3M	Select all contacts. Start conversation.		Conversation screen is displayed. IM editor is empty.

Alternative Flows

From Step: UC_01_1M
To Step: END

Step Id	User Action	System State	System Response
UC_01_1A	Go to Saved Conversations folder.		Saved Conversations folder is displayed.

UC 02 - IM message that contains a phone number structured data

Related requirement(s)

Requirements Codes

Description

This use case starts with a conversation in which there is an IM message with at least an embedded phone number.

Main Flow

From Step: UC_01_3M
To Step: END

Step Id	User Action	System State	System Response
UC_02_1M	Send IM message. IM message contains at least 1 embedded phone number.		Phone is in Conversation Screen.
UC_02_2M	Press Options softkey.	IM editor is not empty. Video Calling is available.	Send Message To option is displayed. Video Call option is displayed.
UC_02_3M	Select Send Message To option.		Multiple List Picker screen is displayed.
UC_02_4M	Select at least 1 phone number.	Unified Messaging Composer is flexed on.	Unified Messaging Composer is displayed.
UC_02_5M	Send message.		Message is sent.

Alternative Flows

From Step: UC_02_1M
To Step: UC_02_5M

Step Id	User Action	System State	System Response
UC_02_1A	Highlight phone number in IM message.	IM editor is empty.	Phone number is highlighted.
UC_02_2A	Press Center Select key.		Send Message option is displayed.
UC_02_3A	Select Send Message option.	Unified Messaging Composer is flexed on.	Unified Messaging Composer is displayed.

From Step: UC_02_2M
To Step: END

Step Id	User Action	System State	System Response
UC_02_4A	Select Video Call option.		Multiple List Picker screen is displayed.
UC_02_5A	Select phone number.		Video Calling screen is displayed.

From Step: UC_02_3M
To Step: END

Step Id	User Action	System State	System Response
UC_02_6A	Select at least 1 phone number.	Unified Messaging Composer is flexed off.	Phone is in Message Type screen.

B.2. User view document generated by CSP 2 CNL tool

Feature 11166

Use Cases

UC_01 – IM Main Functionalities

Related requirement(s)

Requirements Codes
TRS 14293

Description

This use case starts a conversation with a contact in the contact list.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Start IM application.		IM application is displayed.
UC_01_2M	Log in IM Server.		User is logged in.
UC_01_3M	Select all contacts. Start conversation.		Conversation screen is displayed. IM editor is empty.

Alternative Flows

From Step: UC_01_1M
To Step: END

Step Id	User Action	System State	System Response
UC_01_1A	Go to Saved Conversations folder.		Saved Conversations folder is displayed.

UC_02 – IM message that contains a phone number structured data

Related requirement(s)

Requirements Codes

Description

This use case starts with a conversation in which there is an IM message with at least an embedded phone number.

Main Flow

From Step: UC_01_3M
To Step: END

Step Id	User Action	System State	System Response
UC_02_1M	Send IM message. IM message contains at least 1 embedded phone number.		Phone is in Conversation Screen.
UC_02_2M	Press Options softkey.	IM editor is not empty. Video Calling is available.	Send Message To option is displayed. Video Call option is displayed.
UC_02_3M	Select Send Message To option.		Multiple List Picker screen is displayed.
UC_02_4M	Select at least 1 phone number.	Unified Messaging Composer is flexed on.	Unified Messaging Composer is displayed.
UC_02_5M	Send message.		Message is sent.

Alternative Flows

From Step: UC_02_1M
To Step: UC_02_5M

Step Id	User Action	System State	System Response
UC_02_1A	Highlight phone number in IM message.	IM editor is empty.	Phone number is highlighted.
UC_02_2A	Press Center Select key.		Send Message option is displayed.
UC_02_3A	Select Send Message option.	Unified Messaging Composer is flexed on.	Unified Messaging Composer is displayed.

From Step: UC_02_2M
To Step: END

Step Id	User Action	System State	System Response
UC_02_4A	Select Video Call option.		Multiple List Picker screen is displayed.
UC_02_5A	Select phone number.		Video Calling screen is displayed.

From Step: UC_02_3M
To Step: END

Step Id	User Action	System State	System Response
UC_02_6A	Select at least 1 phone number.	Unified Messaging Composer is flexed off.	Phone is in Message Type screen.

B.3. Motorola's component view document

Feature - 12898

Use Cases

UC 02 - Incoming message is opened and moved to the Important Messages folder

Main Flow

From Flow: START
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1M	User	Read incoming message.		Message App
UC_02_2M	Message App	Open incoming message.		Message Viewer
UC_02_3M	User	Open menu.		Message App
UC_02_4M	Message App	Display menu.		Menu Controller
UC_02_5M	Menu Controller	Move to Important Messages option is displayed.		User
UC_02_6M	User	Select the Move to Important Messages option.		Message App
UC_02_7M	Message App	Select the Move to Important Messages option.		Menu Controller
UC_02_8M	Menu Controller	Save message to Important Messages folder.	Message storage is not full	Message Storage App
UC_02_9M	Message Storage App	Message moved to Important Message folder is displayed.		User
UC_02_10M	User	Wait for at most 2 seconds.		User
UC_02_11M	Message App	Next inbox message is highlighted.		List App
UC_02_12M	List App	Available message is selected.		User

Exception Flows

From Flow: UC_02_4M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1E	Message App	Display menu.	Important Message flex is off.	Menu Controller
UC_02_2E	Menu Controller	Move to Important Messages option is not displayed.		User

From Flow: UC_02_8M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_3E	Menu Controller	Save selected message to Important Messages folder.	Message storage is full.	Message Storage App
UC_02_4E	Message Storage App	Display Memory full message.		Display App
UC_02_5E	Message App	Display Clean up dialog.		Display App
UC_02_6E	Display App	Clean up request is displayed.		User
UC_02_7E	User	Perform clean up.		Message App
UC_02_8E	Message App	Display messages for detetion.		List App
UC_02_9E	List App	Message list is displayed for deletion.		User
UC_02_10E	User	Select all messages. Delete messages.		Message App
UC_02_11E	Message App	Delete messages.		Message Storage App
UC_02_12E	Message App	Display Message moved message.		Display App
UC_02_13E	Display App	Message moved to Important Message folder is displayed.		User
UC_02_14E	Message App	Return to inbox folder.		Navigation App
UC_02_15E	Navigation App	System returns to inbox folder.		User

UC 04 - Clean up and Important Messages folder

Main Flow

From Step: START
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_1M	User	Go to Message Center.		Navigation App
UC_04_2M	Navigation App	Display Message Center screen.	Important Message flex is on.	Message App
UC_04_3M	Message App	Important messages folder is displayed.		User
UC_04_4M	User	Open menu		Message App
UC_04_5M	Message App	Display menu		Menu Controller
UC_04_6M	Menu Controller	Clean up Messages option is displayed.		User
UC_04_7M	User	Select Clean up Messages option.		Message App
UC_04_8M	Message App	Scroll to Clean up Messages option. Select option.	Important Message folder contains at least 1 message.	Menu Controller
UC_04_9M	Menu Controller	Important Messages item is displayed.		User
UC_04_10M	User	Select Important Messages item.		Message App
UC_04_11M	Message App	Scroll to Important Messages item. Select item.		Menu Controller
UC_04_12M	Menu Controller	Message types list is displayed.		User
UC_04_13M	User	Select message type.		Message App
UC_04_14M	Message App	Scroll to message type. Select message type.		Menu Controller
UC_04_15M	Menu Controller	Display Clean up Important Messages dialog.		Dialog App
UC_04_16M	Dialog App	Clean up Important Messages dialog is displayed.		User
UC_04_17M	User	Confirm clean up operation.		Message App

UC_04_18M	Message App	Confirm operation		Menu Controller
UC_04_19M	Menu Controller	Confirm operation		Dialog App
UC_04_20M	Dialog App	Transient notice is displayed.		User
UC_04_21M	Dialog App	Display Messages Deleted message		Display App
UC_04_22M	Display App	Messages Deleted transient is displayed.		User
UC_04_23M	Message App	Return to Message Center		Navigation App
UC_04_24M	Navigation App	Message Center folder is displayed		User

Exception Flows

From Step: UC_04_8M
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_25E	Message App	Scroll to Clean up Messages option. Select option.	Important Message folder contains at least 1 message.	Menu Controller
UC_04_26E	Menu Controller	Important Messages item is not displayed.	Important Message folder is empty.	User

From Step: UC_04_17M
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_27E	User	Cancel clean up operation.		Message App
UC_04_28E	Message App	Open Clean up messages screen is displayed.		Navigation App
UC_04_29E	Navigation App	Clean up screen is displayed.		User

B.4. Document generated by CSP 2 CNL tool

Feature – 12898

UC_02 - Incoming message is opened and moved to the Important Messages folder

Main Flow

From Flow: START
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1M	User	Read incoming message.		Message App
UC_02_2M	Message App	Open incoming message.		Message Viewer
UC_02_3M	User	Open menu.		Message App
UC_02_4M	Message App	Display menu.		Menu Controller
UC_02_5M	Menu Controller	Move to Important Messages option is displayed.		User
UC_02_6M	User	Select the Move to Important Messages option.		Message App
UC_02_7M	Message App	Select the Move to Important Messages option.		Menu Controller
UC_02_8M	Menu Controller	Save message to Important Messages folder.	Message storage is not full	Message Storage App
UC_02_9M	Message Storage App	Message moved to Important Message folder is displayed.		User
UC_02_10M	User	Wait for at most 2 seconds.		User
UC_02_11M	Message App	Next inbox message is highlighted.		List App
UC_02_12M	List App	Available message is selected.		User

Exception Flows

From Flow: UC_02_4M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_1E	Message App	Display menu.	Important Message flex is off.	Menu Controller

UC_02_2E	Menu Controller	Move to Important Messages option is not displayed.		User

From Flow: UC_02_8M
To Flow: END

Step	Sender	Message	System State	Receiver
UC_02_3E	Menu Controller	Save selected message to Important Messages folder.	Message storage is full.	Message Storage App
UC_02_4E	Message Storage App	Display Memory full message.		Display App
UC_02_5E	Message App	Display Clean up dialog.		Display App
UC_02_6E	Display App	Clean up request is displayed.		User
UC_02_7E	User	Perform clean up.		Message App
UC_02_8E	Message App	Display messages for detetion.		List App
UC_02_9E	List App	Message list is displayed for deletion.		User
UC_02_10E	User	Select all messages. Delete messages.		Message App
UC_02_11E	Message App	Delete messages.		Message Storage App
UC_02_12E	Message App	Display Message moved message.		Display App
UC_02_13E	Display App	Message moved to Important Message folder is displayed.		User
UC_02_14E	Message App	Return to inbox folder.		Navigation App
UC_02_15E	Navigation App	System returns to inbox folder.		User

UC_04 – Clean up and Important Messages folder

Main Flow

From Step: START
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_1M	User	Go to Message Center.		Navigation App
UC_04_2M	Navigation App	Display Message Center screen.	Important Message flex is on.	Message App

UC_04_3M	Message App	Important messages folder is displayed.		User
UC_04_4M	User	Open menu		Message App
UC_04_5M	Message App	Display menu		Menu Controller
UC_04_6M	Menu Controller	Clean up Messages option is displayed.		User
UC_04_7M	User	Select Clean up Messages option.		Message App
UC_04_8M	Message App	Scroll to Clean up Messages option. Select option.	Important Message folder contains at least 1 message.	Menu Controller
UC_04_9M	Menu Controller	Important Messages item is displayed.		User
UC_04_10M	User	Select Important Messages item.		Message App
UC_04_11M	Message App	Scroll to Important Messages item. Select item.		Menu Controller
UC_04_12M	Menu Controller	Message types list is displayed.		User
UC_04_13M	User	Select message type.		Message App
UC_04_14M	Message App	Scroll to message type. Select message type.		Menu Controller
UC_04_15M	Menu Controller	Display Clean up Important Messages dialog.		Dialog App
UC_04_16M	Dialog App	Clean up Important Messages dialog is displayed.		User
UC_04_17M	User	Confirm clean up operation.		Message App
UC_04_18M	Message App	Confirm operation		Menu Controller
UC_04_19M	Menu Controller	Confirm operation		Dialog App
UC_04_20M	Dialog App	Transient notice is displayed.		User
UC_04_21M	Dialog App	Display Messages Deleted message		Display App
UC_04_22M	Display App	Messages Deleted transient is displayed.		User
UC_04_23M	Message App	Return to Message Center		Navigation App
UC_04_24M	Navigation App	Message Center folder is displayed		User

Exception Flows

From Step: UC_04_8M
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_25E	Message App	Scroll to Clean up Messages option. Select option.	Important Message folder contains at least 1 message.	Menu Controller
UC_04_26E	Menu Controller	Important Messages item is not displayed.	Important Message folder is empty.	User

From Step: UC_04_17M
To Step: END

Step	Sender	Message	System State	Receiver
UC_04_27E	User	Cancel clean up operation.		Message App
UC_04_28E	Message App	Open Clean up messages screen is displayed.		Navigation App
UC_04_29E	Navigation App	Clean up screen is displayed.		User