A Tool to Translate CSP Models into English Requirements

Glaucia Peres
Federal University of Pernambuco
Center of Informatics
P.O. Box 7851, Cidade Universitária
CEP 50732-970 Recife, PE Brazil
gbp@cin.ufpe.br

Alexandre Mota
Federal University of Pernambuco
Center of Informatics
P.O. Box 7851, Cidade Universitária
CEP 50732-970 Recife, PE Brazil
acm@cin.ufpe.br

ABSTRACT
It is well-known from Software Engineering that requirement documents change during development. As requirements change, all artifacts must change as well, including tests. In industry, it is common requirement documents become out-of-date with respect to test cases and implementation information because when test cases and implementation are changed, this does not necessarily update the requirements due to market pressures to deliver the software as soon as possible. This is one of the reasons nowadays the use of formal models is increasing in industry, particularly to represent requirements and their related documents. The direct benefit is capturing these artifacts formally, making them consistent. And to widespread the use of formal methods, they are being used in a hidden way. In this paper such a way means using a controlled natural language---CNL (a subset of English). Therefore, our goal in this paper is to present a tool to translate CSP formal models into CNL documents automatically which will be used in the Motorola CIn/BTC research project.

Categories and Subject Descriptors
D.2 [**SOFTWARE ENGINEERING**]: Miscellaneous;
D.2.1 [**Requirements/Specifications**]: Tools

General Terms
Verification

Keywords
Controlled Natural Language, CSP, Formal Specifications, Software testing

1. INTRODUCTION
It is well-known from Software Engineering that requirements change during development. Therefore, throughout the software life cycle, requirements documents and their related documents, such as test artifacts, always need to be maintained. Tracking changes properly can save time and money, by identifying and fixing potential problems early in the development cycle [3]. When requirements change, it usually means that the system project and the implementation also have to be modified, and that the system has to be tested again. Thus, the challenge is to keep all these artifacts consistent with these new changes.

Nowadays, the use of formal models, which are an abstract way to specify computer systems, is increasing in industry to represent requirements and their related documents. 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 must be avoided. Investing in good requirements specification methodologies is an effective way to reduce costs. The formalism used in our work is the process algebra CSP [7] (Section 2.2).

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, or simply CNL [8] (Section 2.1). They contain a smaller and restricted grammar than the natural languages. Thus, they prevent the writer from introducing ambiguous and non-uniform sentences.

This work has been developed in the context of the CIn/BTC research project, which is sponsored by Motorola Inc. Figure 1 shows the research project initiatives that aim to improve the software testing process through automation.

One of the research project's goals is to automatically update requirements documents from test cases. Some efforts have been made to achieve this objective. We already have a strategy, (1) in Figure 1, to transform CNL requirements into CSP (intermediate representation) [1], another one (3) to transform test cases into CSP (intermediate representation) [10] and another that unifies (updates) CSP models [10]. In the direction of providing the update of requirements with test cases, here we present a tool that translates requirements documents written in CSP into CNL requirements documents. This effort corresponds to the task (6) in Figure 1.

Therefore, the main contribution of our work is a tool to translate CSP models into requirements documents written in an English CNL (Section 3).

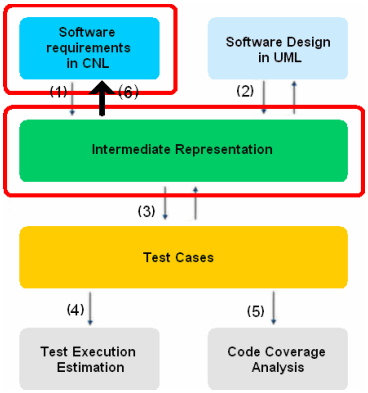

Figure 1 Motorola's research project overview

2. BACKGROUND

In this section, we present a brief overview about the Controlled Natural Language and the CSP specification language.

2.1 The Controlled Natural Language

Controlled Natural Language (CNL) is a processable version of English. The CNL grammar is a subset of the English grammar. Its sentences contain domain specific verbs, terms, and modifiers. The phrases are centered on the verb. Domain terms and modifiers are combined in order to take thematic roles around the verb [2]. This strategy is detailed in [5] where it has been used to translate test cases sentences into CSP constructions. Figure 2 presents a simplified view of the syntax of the CNL. Thus a requirements document is seen as a set of CNL sentences.

```
sentence ::= verb nounPhrase
nounPhrase ::= preposition article modifier noun
      modifier
                  | article modifier noun modifier
                  | modifier noun modifier
                  | modifier noun
                  | noun modifier
                  | noun
verb ::= send | call | select ...
article ::= the | a | an
preposition ::= from | to ...
modifier ::= at least | protected ...
noun ::= phone | folder ...
```

Figure 2 BNF for the Controlled Natural Language

In Figure 3, we observe an example of a use case where the fields `User Action` and `System Response` are written in CNL. The example is a template of a use case document proposed by [1] and it is used to represent the use cases involved in our approach. There are two types of flows: the `Main Flow` and the `Alternative Flows`; for each flow we have the step description (represented by the field `User Action`), the `System State` that complements the steps and the `System Response`. Each alternative flow is related with some step of the `Main Flow`, by the field `From Step`. The field `Step Id` is used to make each step unique in the use case. In our example, the step `UC_01_1M` represents the first step of the use case. The field `To Step` indicates which step follows the last step of its corresponding flow. For instance, the `To Step` of the alternative flow in Figure 3, filled with `SKIP`, says that after its last and single step (`UC_01_1A`) the execution of the flow is over.

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 3 Example of a use case written in CNL

As it is illustrated in Figure 4, the `Start IM Application` is an imperative sentence that has the verb `Start` and the term `IM Application`. In this case, thematic roles are defined to the verb `Start` in order to determine how the verb `to start` is associated with terms. Thus the positions of the terms in the sentence are also defined by the thematic roles. The same happens to the sentence `IM Application is displayed`, thematic roles are defined to the verb `to be` in order to validate the construction `<term> <verb> <modifier>`.

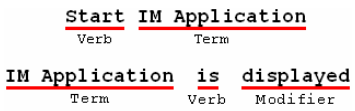

Figure 4 CNL sentences components

2.2 CSP Overview

The term CSP stands for Communicating Sequential Processes and it is defined as a formal language for describing patterns of interaction in concurrent systems [7]. CSP allows the description of systems in terms of component processes that operate independently, and interact with each other using actions (events). The machine-readable version of CSP, CSP_M, enables the use of CSP in practice. Thus, all CSP elements used in this paper is in the CSP_M version.

Each CSP process P uses a set of events called its alphabet, represented as αP. The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators.

Figure 5 shows a fragment of a CSP file, and it will introduce the CSP syntax and semantics. It presents the formal specification of the use case main flow described in Figure 3. The CSP process starts with the description of the main process `System`. It

represents a sequential composition (;) between the process UC_01_1M and itself. This means that System starts behaving like UC_01_1M and will behave like itself (recursively) only when UC_01_1M terminates successfully.

```
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 =
steps -> start.DTSTA_APPLICATION.(IM_APPLICATION,{})->
expectedResults -> isstate.DTISS_APPLICATION_STATEVALUE.
        (IM_APPLICATION,{}).(DISPLAYED_VALUE,{})->
(UC_01_2M [] UC_01_1A)
UC_01_2M =
steps -> login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION,{})->
expectedResults -> isstate.DTISS_USER_STATEVALUE.(USER,{}).
        (LOGGED_IN_STATE,{})->
UC_01_3M
UC_01_3M =
steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM,{ALL})->
start.DTSTA_LISTITEM.(CONVERSATION_ITEM,{})->
expectedResults -> isstate.DTISS_SCREEN_STATEVALUE.
        (CONVERSATION_SCREEN,{}).(DISPLAYED_VALUE,{})->
isstate.DTISS_FIELD_STATEVALUE.(IM_EDITOR_FIELD,{}).
        (EMPTY_STATE,{})->
SKIP
```

Figure 5 CSP file example

A prefix (x -> P), where x is an event and P is a process, represents the process that waits indefinitely by x, and then behaves like the process P. In our example, the event isstate.DTISS_USER_STATEVALUE.(USER.{}).(LOGGED_IN_STATE,{}) is offered to the environment and, if it accepts the event, then the event occurs and the process behaves like UC_01_3M.

The external choice (also known as deterministic choice) determines a choice between two processes and it is up to the environment deciding which one to engage. In Figure 5, (UC_01_2M [] UC_01_1A) means that the environment decides whether UC_01_2M or UC_01_1A will be chosen.

In the CSP Algebra, there are two primitive processes: STOP and SKIP. STOP doesn't communicate anything and it is used to describe the break of a system, as well as a deadlock. SKIP indicates that the execution was contained with success.

A CSP specification can be described using three complementary semantic models: traces, failures and failures-divergences [7]. In our work, we are concerned with the traces model, which describes the sequences of events a process can perform.

3. THE CSP2CNL TOOL

The CSP2CNL tool translates use cases documents written in the CSP notation into their corresponding Microsoft Word 2003 [4] documents, written in the English CNL. That is, it transforms formal specifications into English documents.

The CSP2CNL tool has been developed to be part of a major tool, called TaRGeT. TaRGeT or Test and Requirements Generation Tool is also related to Motorola's context, and it is being developed by the Research Team. The TaRGeT tool as suggested, gives support to automatic generation of test cases and requirements. TaRGeT automates a systematic approach for dealing with requirements, design and test artifacts in an integrated way. Test cases can be automatically generated from both use cases written in natural language and UML sequence diagrams. Moreover, existing test cases can be used to automatically generate or indicate necessary updates to requirements documents.

In our approach, the use cases documents can be written in two different templates defined by [1]. Thus, the use cases can be specified as user view use cases or component view use cases. The main difference between a user 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. Thus, components can share resources and exchange messages.

Both user and component view use cases are translated to their formal model specifications in CSP by the Use Model Generator tool proposed in [1]. The use cases are first written according to the CNL grammar before the translation to their corresponding CSP models. Once the CSP models are created, they become the input to the TCRev tool[10], as can be seen in Figure 6. Then, the TCRev tool creates automatic test cases from the input CSP models or it creates updated CSP models that will be our CSP2CNL tool input files in order to generate the English user or component view use cases documents.

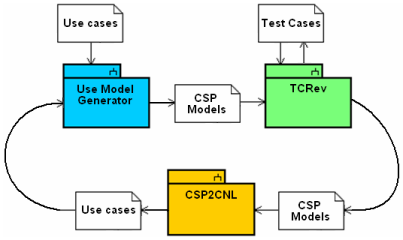

Figure 6 CSP2CNL tool's scenario

Our CSP2CNL tool is not yet working attached to the Use Model Generator and nor to the TCRev. And neither the three of them is yet attached to the TaRGeT. But the idea is that they will be accessed from the TaRGeT's main menu, as it is illustrated in Figure 7. Then, the user chooses any updated test artifact. After that, the artifact will be transformed into its CSP model, which is the intermediate representation to assure the consistency between artifacts as well as generate new artifacts automatically; this CSP model is never seen by the end user (following the philosophy of hidden formal methods). And finally, our tool receives the CSP

model and generates its corresponding use case document in English.

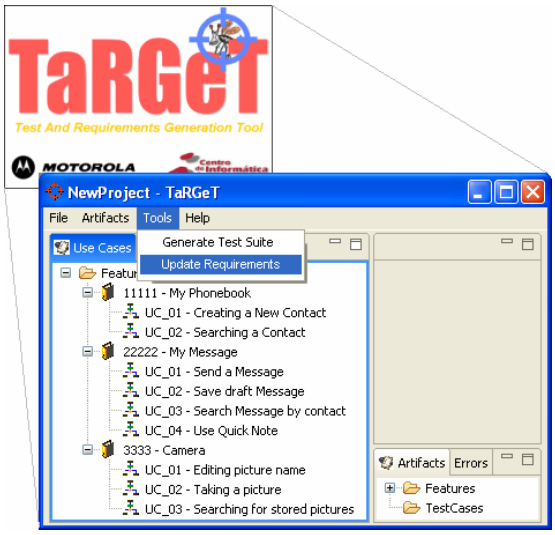

Figure 7 TaRGeT's menu for the CSP2CNL tool

Just for illustrating in this paper the scenario we are trying to explain, after the user chooses the `Update Requirements` option, the TaRGeT will open the `Test Cases` tab, showing all updated test artifacts. Thus, in order to update their corresponding requirements documents, first it is necessary to choose a single test suite, and to right click over it, as it is showed in Figure 8. Then, after that, the user should click in the `Update Requirements` option. Finally, the tool generates the use case document corresponding to the selected test cases. As can be noticed, the user does not need to deal with the CSP models generated during the updating process.

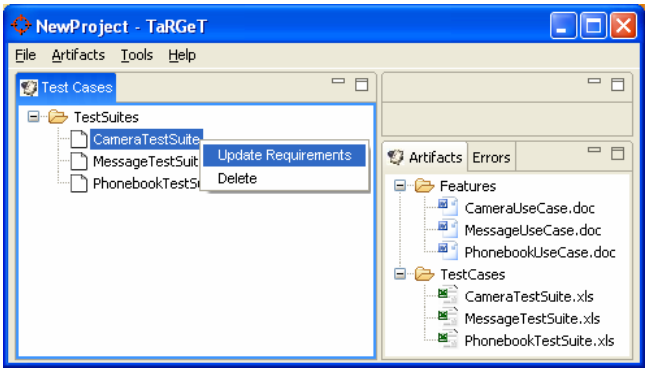

Figure 8 Updating requirements from test cases

Due to the fact that either the user or component view use cases represent different views of the system requirements, they also have distinct CNL template documents representations. The Use Model Generator receives each template document and creates different CSP model structures that will be the input files to the TCRev tool to generate their corresponding automatic test cases documents and the test cases' CSP models. So, each model has particularities that have to be treated separately in order to generate their corresponding use cases documents. Thus, our CSP2CNL gives a different treatment to both user and component view CSP models generated. The following subsections summarize our tools' functionalities.

3.1 User View Use Case Generation

The CSP2CNL tool reads the input CSP file, which comes from the TCRev tool, in order to identify whether it is a user or a 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 3.2. 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 corresponding 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, 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`. Taking the event `select.DTSEL_LISTITEM (CONTACT_ITEM,{ALL})` in step `UC_01_3M` as an example in Figure 5, the CNL generation is done as follows, using the adopted knowledge base.

- 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, as it is illustrated in Figure 3. This functionality is summarized in Section 3.3.

3.2 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 [1].

Figure 9 shows an example of a fragment of a component CSP file. One of the differences that can be noticed is that the events 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 9 is illustrating only the USER_P process, which groups all the messages exchanged between the component 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 as the USER_P process.

```
include "CSP HEADER USER.csp"
System = USER P; System
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 the menu.
 openComp.USER.MESSAGE APP.DTOPE MENU.(CSM MENU LIST, {}) ->
 -- 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 the
 --       "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 9 Component view CSP file

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 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 event name (readComp) and the restriction name (DTREA_SENDABLEITEM) comes the sender (USER) and receiver (MESSAGE_APP) components.

As it was mentioned in Section 3.1, the CSP2CNL tool reads the input CSP file that comes from the TCRev tool, in order to identify whether it is a user or a component view representation. It is done by checking via a regular expression if the term Comp is presented inside the file. Next, just like what happens in the user view use case generation, the tool executes the file parser, keeping in Java objects the information about its use case(s). 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. Taking the event readComp.USER.MESSAGE_APP.DTREA_SENDABLE ITEM.(INCOMING_MESSAGE,{}) in Figure 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.

In order to organize the steps into their corresponding flows, it is necessary to group all messages by their components. Figure 10 illustrates only the messages where the USER component acts as a sender or receiver. As can be noticed, not every step involved in the use case example can be recovered using only the USER_P process described in Figure 9. This means that, to retrieve the whole use case description it is necessary to first read every single process in the complete CSP file, and then assemble the flows by their corresponding steps. Each step that ends with the letter M belongs to the use case main flow, for instance the step UC_02_1M. The steps ending with the letter E belongs to an alternative flow.

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 10 Component view use case example

As the use cases, flows and steps were organized in objects, finally the tool is apt to structure the Word 2003 document, summarized in the next Section. Figure 10 illustrates the

component view use case main flow generated by the CSP file showed in Figure 9.

3.3 Word 2003 Document Generation

In order to write user and component view use cases into Word documents, our tool utilizes both use cases templates defined in Chapter 3 of [1]. Our generation strategy is to get the XML Word representation for both templates and work with them. This representation is called WordprocessingML [6], also known as WordML, which is the XML file format for Microsoft Word 2003 documents. These documents are then used to create templates with Velocity [11], 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 3.1 and 3.2, the Word document generation can be achieved.

After generating with success the updated use cases Word documents, the message in Figure 11 is displayed.

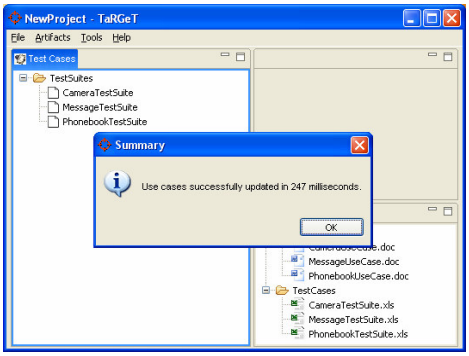

Figure 11 Use cases successfully generated message

4. CONCLUSIONS

Even though the idea that something has to change, most of time, brings apprehension, changing requirements needs to stop to be seen as a threat, to 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 one 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 in 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. 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. It seems to be a great catch to automate the requirements maintenance so that 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.

The automation strategy presented in our work and implemented by the tool CSP2CNL is the link that was missing between the tools developed by [1] and [10] so they could work together. Thus, as a future research, it is necessary that the CSP models files generated by [1] and the ones generated by [10] are created following the same format. Currently, as these projects were developed separately, the CSP models created by them are structured in different formats. Another future work is to make it possible that our CSP2CNL tool works attached to the tools developed by [1], by [10] and the TaRGeT tool, so that the transformation from test cases to use cases could be executed automatically, without users' interaction. Thus, the use cases documents could be always updated at the moment new changes are assigned to their corresponding test cases.

5. REFERENCES

[1] Cabral, G. F. L. *Geração de Especificação Formal de Sistemas a partir de Documento de Requisitos.* Master's thesis, Universidade Federal de Pernambuco, Recife, PE, Brasil, 2006.

[2] Fillmore, C. J. *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, Vol. 208: 20-32, 1976.

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

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

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

[6] Microsoft Corporation *Overview of WordprocessingML.* Redmond, WA, USA, 2003.

[7] Roscoe, A. W., Hoare, C. A. R. and Bird, R. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[8] Schwitter, R., Ljungberg, A. and Hood D. *ECOLE - a look-ahead editor for a controlled language, in: Controlled translation*. In Proceedings of EAMT-CLAW03, May 15-17, Dublin City University, Ireland, 2003.

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

[10] Sousa, C. F. *Modelling and Integrating Formal models: from Test Cases and Requirements Models*. Master's thesis, Universidade Federal de Pernambuco, Recife, PE, Brasil, 2006.

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