ELSEVIER

Electronic Notes in Theoretical Computer Science 195 (2008) 171–188

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Formal Specification Generation from Requirement Documents

Gustavo Cabral [1]

Centro de Informática - CIn
Universidade Federal de Pernambuco - UFPE
Recife, Brazil

and

Mobile Devices R&D Motorola Industrial Ltda
Rod SP 340 - Km 128,7 A - 13820 000
Jaguariuna, Brazil

Augusto Sampaio [2]

Centro de Informática - CIn
Universidade Federal de Pernambuco - UFPE
Recife, Brazil

Abstract

Automatic generation of formal specifications from requirement reduces cost and complexity of formal models creation. Thus, the generated formal model brings the possibility to carry out system property verification. This paper proposes a Controlled Natural Language (CNL, a subset of English), use case specification templates, and a strategy and tool support to generate process algebraic formal models (in CSP notation) from use cases specified using the templates and CNL. We define templates that represent requirement at different levels of abstraction. Moreover, a refinement notion is defined based on events mapping between abstract and concrete models.

Keywords: Use Case Specification, Controlled Natural Language, Formal Specification Generation, Formal Models Refinement, CSP

1 Introduction

Formal methods provide the mathematical basis for achieving software correctness. Nevertheless, its wide adoption in practice is still a big challenge. One of the difficulties faced by the practical software engineer is precisely the cost and complexity [8] involved during system formal specification. These tasks must be cost-effective

[1] Email:gflc@cin.ufpe.br

[2] Email:acas@cin.ufpe.br

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2007.08.032

so that real projects can take advantage of several formal specification benefits [1], such as mechanically analyzing a system to check for deadlock and livelock freedom, among other useful properties.

Rather than building specifications in an *ad hoc* way, some approaches in the literature have explored the derivation of formal specifications from requirements. ECOLE [15] is a look-ahead editor for a controlled language called PENG (*Processable English*), which defines a mapping between English and First-Order Logic in order to verify requirements consistency. A similar initiative is the ACE (Attempto Controlled English) project [5] also involved with natural language processing for specification validation through logic analysis. The work reported in [6] it is established a mapping between English specifications and finite state machine models. In industry, companies, such as Boeing [18], use a controlled natural language to write manuals and system specifications, improving document quality. There are also approaches that use natural language to specify system requirements and automatically generate formal specifications in an object-oriented notation [9].

We propose a strategy that automatically translates use cases, written in a Controlled Natural Language, into specification in CSP process algebra [13]. For obvious reasons, it is not possible to allow a full natural language as a source. We define a subset of English, which we call Controlled Natural Language (CNL), with a fixed grammar, in order to allow the mechanized translation into CSP.

The context of this work is a research cooperation between CIn-UFPE and Motorola called CInBTCRD. Therefore, the proposed CNL reflects this domain. The formal model generated in CSP is used as input by other tools developed in this project. This model is internally used by these tools to automatically generate test cases, both in Java (for automated ones) and CNL itself (for manual ones).

Unlike the cited approaches, which focus on translation at a single level, we consider use case views possibly reflecting different levels of abstraction of the application specification. This is illustrated in this paper through a user and a component view. We also explore a refinement relation between these views; the use of CSP is particularly relevant in this context: its semantic models and refinement notions allow precisely capturing formal relation between user and component views. The approach is entirely supported by tools. A *plug-in* to Microsoft Word 2003 [17] has been implemented to allow checking adherence of the use case specifications to CNL grammar. Another tool has been developed to automate the translation of use cases written in CNL into CSP. Finally, FDR [12], a CSP model checker, is used to check refinement between user and component views.

Section 2 gives an overview of the proposed approach. In Section 3 we present the templates defined to write use cases in CNL and tools that ensure its correct usage. Section 4 focuses on the translation from use cases in CNL to models in CSP. In Section 5 we explore a refinement relation between CSP models describing user and component views and how this can be mechanically checked using FDR. Section 6 summaries our contributions and suggest topics for further research.

2 Strategy Overview

In our approach, each use case is specified using a template. The template is structured to hold information concerning traceability with requirements, a brief description and the way actor interacts with the system. There are two use case templates: the *user view* and the *component view*.

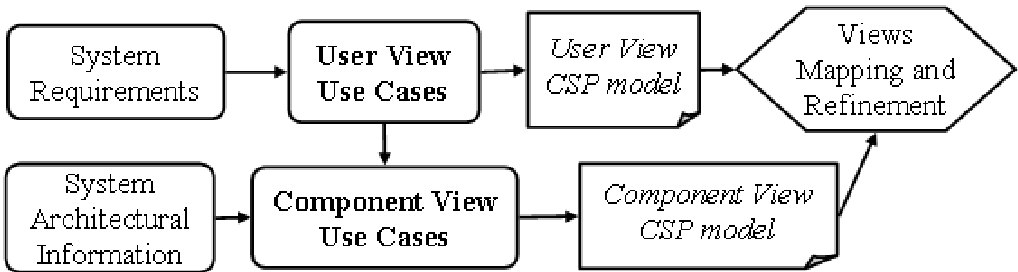

Fig. 1. Proposed strategy overall process

As shown in Figure 1, after `System Requirements` are described in an abstract way, defining what the system is intended to perform, **user view use cases** are created based on requirements analysis. This first set of use cases designs the ways actors interact with the system. Later, **component view use cases** are created based on the **user view use cases** and the adopted `System Architectural Information`, as presented in Figure 1.

The language used to write these use cases is a Controlled Natural Language (CNL), a subset of English relevant to the specific domain. Using CNL it is possible to write imperative and declarative sentences. An imperative sentence describes actor actions and a declarative sentence depict system characteristics, such as a GUI description, or the system state. CNL is necessary to restrict vocabulary used to write use cases; its grammatical rules are defined through knowledge bases that map verbs to CSP channels and verb complements to values of CSP datatypes. Besides aiming at automatic generation of formal models, the use of CNL also prevents the introduction of ambiguous sentences in the use case specification, therefore contributing to the quality of documentation.

Each use case sentence is translated into a CSP event, and a sequence of sentences produces a sequence of CSP events, combined with the CSP *prefix* operator, which gives rise to a CSP process. Each use case defines part of the system formal specification. The presence of alternative or exception execution flows in use cases is captured by the CSP *choice* operator, thus allowing processes combination. Hence, the **user view use cases** are translated into a *user view use model* and the **component view use cases** are translated into a *component view use model*.

Finally, the relation between user and component use cases is established by a mapping from the more abstract to the more concrete model. This event mapping relation is used to prove that the component view model is a refinement of the user view model. The following sections detail the steps of the strategy.

3 Writing Use Cases based on Templates and CNL

Use case specifications capture the system behavior, possibly at different levels of abstraction. In this section we present specifications from the user perspective and from the point of view of system components.

3.1 User View Use Case

User view use cases specify system behavior when one single user executes it. It specifies user operations and expected system responses. Thus, the pair *(user action, system response)* is called a *step*. Every step is identified through an *Id*.

A sequence of possible user actions and system responses is called an *execution flow*. Every execution flow is defined based on a starting point, or *initial state*, and a *final state*. Each execution flow starting point and final state is represented by the *From steps* and the *To step* fields, which represent references between use case execution flows, making it possible to reuse steps and define loops.

There are situations when user can accomplish more than one action given the current system state. When this happens it is necessary to define one execution flow for each of the possible actions. Execution flows are categorized as *main*, *alternative* or *exception* flows, based on their nature. The existence of alternative and exception flows is also related to the system state column. At a given state, the system may respond differently given the same user action. In this case, it must be specified a different system state for each of the possibilities.

UC_02 - Incoming message moved to the Important Messages folder

Related requirement(s): REQ_1302, REQ_1326
Description: User accepts an incoming message and moves it to the Important Messages folder.

Main Flow
From Steps: START
To Step: END

Step Id	User Action	System State	System Response
1M	Read incoming message.		Message content is displayed.
2M	Open the menu.	"Important Messages" feature is on.	"Move to Important Messages" option is displayed.
3M	Select "Move to Important Messages" option.	Message storage is not full.	"Message moved to Important Messages folder" is displayed.
4M	Wait for at most 2 seconds.		The next message is highlighted.

Exceptions Flow
From Steps: 2M
To Step: END

Step Id	User Action	System State	System Response
1E	Select "Move to Important Messages" option.	Message storage is full.	"Memory required" dialog is displayed.
2E	Confirm memory information dialog.		Message content is displayed.

Table 1
Example of a user view use case

Table 1 is a use case example used to specify a functionality presented in most mobile phones. This use case specifies that messages received by the phone can be moved from the inbox to a special folder. The user view use case includes a list of related requirements, a brief description of the use case, and two execution flows: the main and the exception flow. The *From steps* field, in the main flow, is defined as `START` so this flow does not depend on any other flow, and it is a starting point to navigate through the system functionalities. The *To step* field is set as `END` so once the four steps from the main flow are executed the system stops successfully and the user can execute any use case that have the *From steps* set to `START`.

The system state column is mainly used to specify conditional situations. Note that this example captures one exception flow. The normal execution of the main flow would pass through the step `2M`, and go on until the end of the main flow. The exception execution goes from step `2M` to step `1E`, when the `message storage is full` (system state). In this case, given the same `Select Move to Important Messages option` user action, depending on the system state a different system response is presented.

3.2 Component View Use Case

A component view use case specifies the system behavior based on user interaction with system components. In this model, the system is decomposed into components that concurrently process user requests and communicate among themselves. Table 2 shows the component view use case that refines the use case in Table 1.

In the component view it is necessary to define 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 is actually viewed here as a component, and can either send or receive messages to or from other components. A component can also send a message to itself. The idea of execution flows is the same as in the user view and the system state column plays the same role as previously described in the user view.

In Table 2, there is one main and one exception flow. The execution of the main flow can be deviated to an exception path after step `7M`, when the `Message App` sends a message to the `Menu Controller` component. Here, the next message to be exchanged depends on the current system state. Just like in the user view example, the `Message Storage` state (full or not full) determines the next message to be exchanged between the components. Note that the exception flow step `1E` is activated after the step `7M`, when the condition fails. The `To step` field, in the exception flow, states that after the execution flow finishes the execution of the use case terminates (`END`); it could alternatively transfer control back to the main flow.

3.3 Controlled Natural Language

Use case fields (*user action*, *system state*, *system response*, and *message*) are written in a Controlled Natural Language with a fixed grammar, defined by knowledge bases. The following subsections briefly describe these knowledge bases involved in

Main Flow
From Steps: START
To Step: END

Step Id	Sender	Message	System State	Receiver
1M	User	Read incoming message.		Message App
2M	Message App	Open incoming message.		Message Viewer
3M	User	Open the Menu.		Message App
4M	Message App	Display Menu.	“Important Messages” feature is on.	Menu Controller
5M	Menu Controller	“Move to Important Messages” option is displayed.		User
6M	User	Select the “Move to Hot Messages” option.		Message App
7M	Message App	“Move to Important Messages” option.		Menu Controller
8M	Menu Controller	Save message at “Important Messages” folder.	Message storage is not full.	Message Storage App
9M	Message Storage App	“Message moved to Important Messages folder” is displayed.		User
10M	User	Wait for at most 2 seconds.		User
11M	Message App	The next inbox message is highlighted.		List App
12M	List App	Available message is selected.		User

Exception Flow
From Steps: 7M
To Step: END

Step Id	Sender	Message	System Response	Receiver
1E	Menu Controller	Save message at “Important Message” folder.	Message storage is full.	Message Storage App
2E	Message Storage App	“Memory required” message is displayed.		Display App
3E	User	Confirm memory information dialog.		Message App
4E	Message App	Message content is displayed.		User

Table 2
Example of a component view use case

the definition of the CNL [10].

3.3.1 Lexicon

The Lexicon stores vocables that may appear in CNL sentences. Each vocable may be a *verb*, a *term*, or a *modifier*. A verb is used to define an action or the system state. A term is an element, or entity, from the application domain. Finally, a modifier can be an adjective or an adverb.

Figure 2 gives examples of application domain term and modifier definitions. This example defines two terms: `message storage is full`, referring to a dialog name, and `message storage`, referring to an application item that can be manipulated somehow. The modifiers are `only` and `correctly`. Their definitions contain the `position` and `precedence` fields that determine how they are positioned among terms or other modifiers. The `number inflection` defines whether it is a `singular`

or `plural` modifier. The `article` field determines whether the modifier accepts an article or not.

```
<noun>
  <term>message storage</term>
  <plural/>
  <model>MESSAGE_STORAGE</model>
  <class>item</class>
</noun>
<noun>
  <term>message storage is full</term>
  <plural/>
  <model>MESSAGE_STORAGE_FULL</model>
  <class>dialog</class>
</noun>
```

```
<modifier>
  <term>only</term>
  <position>before</position>
  <precedence>0</precedence>
  <numberinflection>singular</numberinflection>
  <article>no</article>
  <model>ONLY</model>
</modifier>
<modifier>
  <term>correctly</term>
  <position>both</position>
  <precedence>1</precedence>
  <numberinflection>plural</numberinflection>
  <article>no</article>
  <model>CORRECTLY</model>
</modifier>
```

Fig. 2. Term and modifier definitions in Lexicon

3.3.2 Ontology

Each application domain has specific elements and entities represented as terms, which are grouped in classes according to their characteristics. These classes are related by inheritance. Figure 3 presents a small fragment of the Ontology that defines the `Object`, the `Value`, and the `State Value` classes. The `State Value` class inherits from the `Value` class, and the `Value` class inherits from the `Object` class. Note that, in Figure 2, the term `message storage is full` is a dialog due to the fact that it belongs to the `dialog` class of the 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:
            "enabled","ON","high".</description>
          <name>State Value</name>
          <code>state_value</code>
          <subclasses/>
        </class>
...
```

Fig. 3. Ontology fragment

3.3.3 Case Frame

The case frame defines the relation between verbs, terms and modifiers. Each case frame determines how a verb can be used to instantiate a sentence. We use the case grammar formalism [4] that contains information about the input domain verbs and its thematic roles, which can be an agent or a theme of the sentence. When

a sentence is constructed, each term, along with modifiers, takes a thematic role around the verb. Each case frame can also be associated to more than one verb, all of them assuming the same meaning. Figure 4 is the definition of the SelectItem case frame, which is defined by two verbs select and choose.

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

Fig. 4. Case frame example

3.4 Case Frame Restriction

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

```
<frame>
 <description>Set the value of an item.
    Example: Set the Fix Dialing to on
 </description>
 <name>SetItem</name>
 <verblist>
   <verb>set</verb>
   <verb>check</verb>
 </verblist>
 <roles>
   <role mandatory="True">agent</role>
   <role mandatory="True">theme</role>
   <role mandatory="false">to-value</role>
 </roles>
</frame>
```

```
<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_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>
```

Fig. 5. Case frame and respective case frame restriction example

Figure 5 contains the case frame definition SetItem for the verbs set and check, and its respective case frame restriction. Observe that this case frame contains the following roles: agent, theme, and to-value. Based on these roles, there are four defined restrictions: the three first restrict the theme and the to-value arguments, and the last one restricts only the theme argument, once the to-value argument is not mandatory. Each restriction has a name; this name 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 example: the `DTSET_FIELDVALUE_FIELD` restriction defines that the `theme` is a term from the `field` class and the `to-value` argument belongs to the `field_value` class.

3.5 Tool Support

Because use case sentences must be adherent to CNL rules, use case designers have to know the CNL grammar. It is a complex task; CNL domain terms and expressions may be constantly updated each time a new set of requirements arrives.

In order to minimize this problem, we have developed a Microsoft Word 2003 [17] *plug-in* that ensures use cases are written according to use case templates and CNL syntax. It enforces the use of the templates, through XML schemas, and verifies each use case sentence; if there are sentences not according to CNL rules it assists the designer to rewrite it.

Two modules compose this tool. One is implemented using the .NET Platform [7] and the other is implemented in Java [19]. The .NET module is a GUI program that accomplishes the CNL validation within Word. The Java module is the Natural Language Processing (NLP) unit responsible to verify if sentences are written according to CNL rules. More details about the NLP module implementation can be found at [10].

4 CSP Specification Generation

Once use cases are created using the tool mentioned in Section 3.5 and follow the proposed templates and CNL, it is possible to generate a CSP model from it.

4.1 CSP Notation

The CSP process algebra [13] is the target formalism of our strategy. CSP allows the description of systems in terms of processes that operate independently, and interact with each other through message-passing communication. The relationship between processes is described using process algebraic operators from which complex process compositions can be constructed from few primitive constructors.

The behavior of a CSP process is described in terms of events, which are atomic and instantaneous operations, such as *open* or *close*, which may transmit information. As an example, the communication $open!door$ outputs the value *door* through the channel *open*. There are two primitive processes: `STOP` and `SKIP`. `STOP` communicates nothing and stands for a canonical deadlock; `SKIP` represents successful termination.

Some of CSP operators are the prefix ($a \rightarrow P$), deterministic choice ($P \Box Q$), nondeterministic choice ($P \sqcap Q$), interleaving ($P|||Q$), the parallel composition ($P[|s|]Q$, where s is the set of events in which P and Q synchronize), and hiding ($P \setminus s$, where s is the set of events to be hidden). The prefix operator combines an event and a process to produce a new process. The deterministic (or external) choice operator allows the future behavior of a process to be defined as a choice between two

component processes. The nondeterministic (or internal) choice operator allows the future evolution of a process to be defined as a choice between two component processes, but does not give the environment any control over which of the component processes are selected. The interleaving operator represents completely independent concurrent activity. The parallel composition (interface parallel) operator represents concurrent activity that requires synchronization between the component processes. The parallel composition operator is also defined as $P[p||q]Q$, where p and q are set of events accepted by the processes P and Q respectively. The hiding operator provides a way to abstract processes, by making some events unobservable.

```
channel a, b, c

events_view_1 = { a, b, c}
View_1 = a -> ( b -> View_1
              [] c -> View_1)
```

```
channel a1, a2, a3, b1, b2, c1

events_view_2 = {a1, a2, a3, b1, b2, c1}
View_2 = a1 -> a2 -> a3 ->
              ( b1 -> b2 -> View_2
              [] c1 -> View_2)
```

Fig. 6. CSP process examples

In Figure 6, the `View_1` and the `View_2` processes are defined. The channels (events) `a`, `b`, and `c` are used by and constitutes the *alphabet* of `View_1`, and the channels `a1`, `a2`, `a3`, `b1`, `b2`, and `c1` are the *alphabet* of `View_2`. Both processes `View_1` and `View_2` use the *prefix* and the *choice* operator. For instance, after engaging in event `a`, `View_1` offers `b` and `c` to the environment. After engaging in `b` or `c` it recurses.

4.2 CSP Events Generation

Based on the presented CNL knowledge bases, we define the CSP alphabet channel names and the datatypes of the model. The verbs determine CSP channel names. Each class from the Ontology defines a CSP datatype. The terms and modifiers from the Lexicon are related to classes from the Ontology and therefore define datatype values. Using these mappings and the case frame definitions, it is possible to translate each sentence from the use cases into CSP events.

Figure 7 presents a sentence from step `3M` in the use case from Table 1 and its translation to a CSP event. The sentence `Message storage is not full` contains the verb `to be` used to describe some `Message storage` characteristic. The verb `to be` is mapped to the event `isstate`. The subject and the predicate from this sentence determine the datatype values used by the `isstate` event: `MESSAGE_STORAGE`, `FULL_STATE_VALUE`, and `NOT`.

```
Message storage is not full.

isstate.DTISS_ITEM_STATEVALUE.(MESSAGE_STORAGE,{}).(FULL_STATE_VALUE,{NOT})
```

Fig. 7. Example of a CNL sentence and its translation to a CSP event

However, mapping CNL sentences to CSP events is just the first step to create the CSP model. The specification generated depends on the use case template. The

following sections explain the generation strategy for the user and the component view use cases.

4.3 User View Model

Each *step* of a use case execution flow is mapped to a CSP process. The process name is defined by the *step id* combined with the use case *id*, forming a unique identifier among all use case steps. This process body contains control events (*steps*, *conditions*, and *expectedResults*) that delimit the events generated from the user action, system state, and system response fields of the use case template (see Table 1).

As already explained, each execution flow has *From steps* and *to step* fields. They determine when the flow starts and ends. They may refer to the steps from other execution flows or to the `START` and `END` keywords.

```
System =
  UC_02_1M ; System
  [] ...

UC_02_1M =
  -- Read incoming message.
  ( steps -> read.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{}) ->
  -- Message content is displayed.
  expectedResults -> display.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE,{}) -> UC_02_2S)

UC_02_2M =
  -- Open the CSM.
  ( steps -> open.DTOPE_MENU.(CSM_MENU_LIST,{}) ->
  -- "Important Message" feature is on.
  conditions -> isstate.DTISL_LIST.(FEATURE,{IMPORTANT_MESSAGE_FOLDER}).(ON_VALUE) ->
  --  "Move to Important Messages" option is displayed.
  expectedResults -> isstate.DTISS_MENUITEM_STATEVALUE.
          (MOVE_TO_HOT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) -> UC_02_3S)

UC_02_3M =
  -- Select the "Move to Important Messages" option.
  ( steps -> select.DTSEL_MENUITEM.(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}) ->
  -- Message storage is not full.
  conditions -> isstate.DTISS_ITEM_STATEVALUE.
          (MESSAGE_STORAGE,{}).(FULL_STATE_VALUE,{NOT}) ->
  -- "Message moved to Important Message folder" is displayed.
  expectedResults -> isstate.DTISS_DIALOG_STATEVALUE.
          (MESSAGE_MOVED_TO_IMPORTANT_MESSAGE_FOLDER,{}).(DISPLAYED_VALUE,{}) ->  UC_02_4M)
  [] UC_02_5E

UC_02_4M =
  -- Wait for at most 2 seconds.
  ( steps -> wait.DTWAI_ITEM.(SECOND, {AT_MOST.2}) ->
  -- The next message is highlighted.
  expectedResults -> isstate.DTISS_SENDABLEITEM_STATEVALUE.
             (MESSAGE,{NEXT}).(HIGHLIGHTED_VALUE,{}) -> SKIP )
```

Fig. 8. Generated CSP specification from the user view use cases of Table 1

Figure 8 shows the generated CSP model for the use case specified in Table 1. It contains the `System` process, which is the main process, and four other processes that refer to steps from the use case main flow. The `System` process refer to the process `UC_02_1M` and any other execution flow with the *From steps* containing `START`. The process `UC_02_3M` is defined as a CSP *choice* between the rest of the main execution flow, the process `UC_02_4M`, and the exception flow, the `UC_02_1E` process. The process `UC_02_4M` is finalized with the `SKIP` process, once the *To step* field is set to `END`.

4.4 Component View Model

The component view model is quite different from the user view one. The component channels contain information about the components involved in the message exchange and its name is suffixed by `Comp`, making the user and component view CSP alphabets different. The datatypes used in both views are the same; since both use cases refer to elements from the same domain.

```
SubSystem1 = USER_P
    [User_Channels||Message_App_Channels]
    MESSAGE_APP_P
SubSystem1_events = union(User_Channels,
                          Message_App_Channels)
```

```
SubSystem2 = SubSystem1
  [SubSystem1_events||Message_View_Channels]
  MESSAGE_VIEWER_P
```

Fig. 9. Part of the component processes composition

In Figure 9, the top level process that represents the component view model is defined by the parallel execution of system components, including the user. They are composed pairwise using the alphabetized parallel operator. Each component accepts a set of events for synchronization; `User_Channels` and `Message_App_Channels` are example of synchronization sets.

Each component has a main process that is defined by *external choice* among the component possible behaviors, depending on the use case. Basically, each use case gives rise to a subprocess for each component, defined by the messages exchanged between itself and other components.

Unlike the user view, each step is mapped into two CSP events, one for each component that takes part in the communication. Each step defines events for the message passed between the components and the system state. After the message itself there is a CSP *prefix* to the next step that involves the component. In Figure 10, it is defined part of the `USER_P` process for one use case. Events `readComp.USER.MESSAGE_APP` and `isstateComp.MENU_CONTROLLER.USER` are examples of the communication between the user and system components.

Similar to the user view, if there are *alternative* or *exception* flows, the *external choice* operator is used to capture the alternatives. In Figure 10, the `USER_UC_02` process contains an *external choice* between the processes `USER_UC_02_9M` and `USER_UC_02_3E` to denote the exception flow.

4.5 Tool Support

A tool has been implemented to mechanize the translation of the user and the component views into CSP models. It reads user and component views use cases as Word 2003 document files, checks its content (invoking the tool presented in Section 3.5), and generates the user and the component models. The NLP module [10] is once again used to translate CNL sentences into CSP events. This application itself accomplishes the arrangement of these events based on the use case structure so the CSP model can be generated.

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

USER_UC_02 =
  -- Message: Read incoming message.
  readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{}) ->
  -- Message: Open the CSM.
  openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST,{}) ->
  -- Message: "Move to Important Messages" option is displayed.
  isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
            (MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) ->
  -- 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 =
  -- Message: "Message moved to Important Message folder" is displayed.
  isstateComp.MESSAGE_STORAGE_APP.USER.DTISS_DIALOG_STATEVALUE.
           (MESSAGE_MOVED_TO_IMPORTANT_MESSAGE_FOLDER,{}}).(DISPLAYED_VALUE,{}) ->
  -- Message: Wait for at most 2 seconds.
  waitComp.USER.USER.DTWAI_ITEM.(SECOND,{AT_MOST.2}) ->
  -- Message: The next inbox message is highlighted.
  isstateComp.USER.LIST_APP.DTISS_SENDABLEITEM_STATEVALUE.
           (INBOX_MESSAGE,{NEXT}).(HIGHLIGHTED_VALUE,{})->
  -- Message: Available message is selected.
  isstateComp.LIST_APP.USER.DTISS_SENDABLEITEM_STATEVALUE.(MESSAGE,{}).
           (AVAILABLE_VALUE,{}) -> USER_P
```

Fig. 10. User process exchanging messages with other components

5 Model Refinement

Modeling systems at different levels of abstraction has the advantage of capturing several architectural views, as illustrated here with the user and the component views. Nevertheless, it is essential that the several architectural views produced are consistent. In general, these views are expressed using different alphabets (event names) so a relation is needed in order to compare them. One or more events from one model can be related to one or more events of another model. Defining a relation allows replacing abstract events with more concrete ones, formally keeping track of the relationship between the models.

5.1 Abstraction Levels

This paper defines only two abstraction levels, user and component views. However, the strategy presented in this section can be generalized for an arbitrary number of views. Use case engineers can define new use case templates and propose new ways to map events from use cases written at different levels of abstraction.

The main goal of our approach is to decompose events using other events that also represent system behavior, in an incremental way. This would enrich the model with more details and eventually the events can be mapped into more concrete constructions, such as programming languages commands (typically method calls).

5.2 Refinement Mapping

Here we consider that the relation between user and component models is a mapping from sequences of user events to sequences of component events. In order to avoid

nondeterministic behavior it should be defined a *one to one* relationship between sequence of events from the two models.

The mapping is defined through a CSP function that receives a list of pairs of sequences and yields a CSP process that represents the mapping. In each pair, the first sequence represents events from the user view, and the second sequence contains events from the component view.

Figure 11 presents the function that generates the mapping process used in the refinement; `MAPPING_FUNCTION` receives the mapping between the two views and use it to create a process using the `MAPPING_PROCESS` function, which is defined as an indexed external choice among the processes generated by the `makeProcess` auxiliary function. This function takes each pair from the mapping and forms a sequence initiated by the events from the abstract model followed by events from the concrete model, terminating by the `SKIP` process.

```
MAPPING_FUNCTION(map) = MAPPING_PROCESS(map);
                        MAPPING_FUNCTION(map)

MAPPING_PROCESS(map) = [] p : map @
             makeProcess(first(p)^second(p))

makeProcess(<>) = SKIP
makeProcess(<a>^as) = a -> makeProcess(as)
```

Fig. 11. Mapping function

The process that represents the mapping is composed, through an alphabetized parallel composition, with the abstract model. This composed process contains events from the alphabet of both views. Once the events from the abstract model are hidden, it produces a process that is refined by the concrete model. The mapping process works as a trigger from one view to another; events executed in the abstract model force the execution of the related concrete events.

The processes `View_1` and `View_2` from Figure 6 are simple examples of abstract and a concrete models. It aims to illustrate the proposed strategy before showing the user and the component view mapping. The `View_1` model is more abstract than `View_2`, and the strategy can be used to replace abstract events from `View_1` with more concrete ones, using `MAPPING_FUNCTION`. Figure 12 presents the *mapping* between the two models and defines the process `View_1_with_mapping`, which have the events from `View_1` hidden, resulting in `View_1_mapped` that is refined by the `View_2` model.

```
map = {(<a>,<a1,a2,a3>),(<b>,<b1,b2>),
       (<c>,<c1>)}

View_1_with_mapping = View_1
      [|events_view_1|] MAPPING_FUNCTION(map)

View_1_mapped = View_1_with_mapping
                 \events_view_1

View_1_mapped [FD= View_2
```

Fig. 12. Mapping function usage example

The last line of Figure 12 captures the assertion that `View_1_mapped` is refined by `View_2` in the failures-divergence model. This mapping strategy is based on a framework composition technique [11]. Here we focus on relating events from

different models, while the framework composition strategy aims to accomplish communication between frameworks possibly with different alphabets.

5.3 Component View as a Refinement of the User View

The idea presented in the previous section can be used to relate user and component view models. In this case the component view model refines the user view through events mapping even though it contains a more complex structure, such as parallel composition.

```
map = { ( < steps, read.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, {}),
expectedResults,display.DTDIS_FIELDVALUE.(MESSAGE_CONTENT_FIELD_VALUE,{})>,

< readComp.USER.MESSAGE_APP.DTREA_SENDABLEITEM.(INCOMING_MESSAGE, {}),
openComp.MESSAGE_APP.MESSAGE_VIEWER.DTOPE_SENDABLE_ITEM. (INCOMING_MESSAGE, {}) > ),

( < steps,open.DTOPE_MENU.(CSM_MENU_LIST, {}),
conditions,isstate.DTISL_LIST.(IMPORTANT_MESSAGES_FOLDER, {}),
expectedResults,isstate.DTISS_MENUITEM_STATEVALUE.
(MOVE_TO_IMPORTANT_MESSAGES_OPTION, {}). (DISPLAYED_VALUE, {}) > ,

< openComp.USER.MESSAGE_APP.DTOPE_MENU.(CSM_MENU_LIST, {}),
displayComp.MESSAGE_APP.MENU_CONTROLLER.DTDIS_MENU.(CSM_MENU_LIST, {}),
isstateComp.MESSAGE_APP.MENU_CONTROLLER.DTISS_FEATURE.(VALUE,{ON}),
isstateComp.MENU_CONTROLLER.USER.DTISS_MENUITEM_STATEVALUE.
(MOVE_TO_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) > ), ... }
```

Fig. 13. Mapping between abstract and concrete views

Figure 13 presents part of the `mapping` between the user and the component view events. The step `1M` from the user view is mapped to the steps `1M` and `2M` from the component view, and the step `2M` is mapped to steps `3M`, `4M`, and `5M`, establishing a relation between user and component views (Figure 14).

```
User_View_with_mapping = User_View
     [| events_user_view |] MAPPING_FUNCTION(map)

User_View_mapped = User_View_with_mapping
                   \events_user_view

User_View_mapped [FD= Component_View
```

Fig. 14. Mapping process specification based on the map

As explained, the component view refines the user view through events mapping. Furthermore, the equivalence between these models, using the proposed approach, can be achieved if an inverse mapping is defined from the component to the user view.

5.4 Tool Support

The generation of the mapping between the user and the component views is automated by the same application that generates the CSP model from the use cases. The use cases, as Word 2003 documents, are read and the mapping is generated based on the user messages in the component view. A sequence of events in the

component view always starts with a user request and ends with a message received by the user. This information is used to map the events from the user to the component view.

The refinement relation discussed here can be mechanically checked using FDR [12], a refinement checker for CSP. After loading the two models and the mapping functions, along with the generated mapping, the only remaining task is to define assertions, such as in Figure 15, to check system properties. The first assertion is related to the illustrative example from Figure 12 and the second is related to the user and component view refinement from Figure 14. The results established that both refinements hold, as expected.

```
assert View_1_mapped [FD= View_2
assert User_View_mapped [FD= Component_View
```

Fig. 15. Assert commands verified by FDR tool

Also based on refinement checking, FDR can verify if a model is deadlock, livelock or nondeterminism free. Moreover, CSP operators bring the possibility to accomplish quite complex compositions and FDR can be used to verify elaborate system properties.

6 Final Considerations

The proposed strategy focuses on generating formal specification through validation and processing of requirements at an early stage. The sooner the requirements are validated, the lower is the risk involved in the system development; problems can be found and analyzed even before system implementation starts. The use of a CNL and use case templates seem relevant to guarantee requirements consistency.

The use of a restricted natural language to write requirements is approached by other works, such as [14], which processes CNL and generate a First-Order Logic. Apart from the fact that we use process algebra as formal model, our strategy goes beyond the translation itself: it generates structured models, possibly at different levels of abstraction, and addresses the formal refinement between them. Furthermore, along with the proposed strategy, there are tools that mechanize the entire process: from the use case specifications creation to the refinement checking. These tools are essential to the introduction of formal methods in real projects, as in the Motorola environment.

When analyzing the user view use cases, it is also possible to retrieve information about the use cases relation. The link between execution flows (*From steps* and *To step*) can be seen as a UML [16] inclusion operator between use cases. Once an execution flow starts from other flow, it includes this latter flow steps in its definition.

Another benefit of this strategy is related to the possible uses of the generated models. The user view model contains important information related to user actions and system responses. This is essential information used to define test cases. There

are several approaches related to Model Based-Testing that use system specifications to generate test cases. In particular, the user view models generated by the presented strategy are used in the CInBTCRD research project to automatically generate test cases based on test purposes [2]. There is also complementary work in the CInBTCRD research project that uses the proposed component view model to generate UML diagrams; in [3] a set of laws is proposed to map CSP specifications into UML-RT diagrams, which is now part of UML 2.0.

The proposed model refinement strategy, through events mapping, and the use case validation approach can also be used as an important step toward automating test case execution. The execution of user actions based on atomic events, associated with automatization of test case verification would enable the execution of test cases generated from the model. Along with code generation, test scripts generation is a possible topic for future investigation associated to our strategy.

7 Acknowledgment

This work has been developed in the context of a research cooperation between Motorola Inc. and CIn-UFPE. We thank the entire group for all the support, criticisms and suggestions throughout this research.

References

[1] Bowen, J. and M. Hinchey, *Seven more myths of formal methods*, IEEE Softw. **12** (1995), pp. 34–41.

[2] Cartaxo, E., "Test Case Generation by means of UML Sequence Diagrams and Label Transition System for Mobile Phone Applications," Master's thesis, Universidade Federal de Campina Grande (UFCG) (2006).

[3] Ferreira, P., A. Sampaio and A. Mota, *Viewing CSP specifications with UML-RT diagrams*, in: *Brazilian Symposium on Formal Methods (SBMF)*, 2006.

[4] Fillmore, C., *Frame semantics and the nature of language*, In Proceeding of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech **280** (1976).

[5] Fuchs, N., U. Schwertel and R. Schwitter, *Attempto Controlled English - not just another logic specification language*, in: *LOPSTR'98: Proceedings of the 8th International Workshop on Logic Programming Synthesis and Transformation* (1990), pp. 1–20.

[6] Holt, A., *Formal verification with natural language specifications: guidelines, experiments and lessons so far*, South African Computer Journal **24** (1999), pp. 253–257.
URL `citeseer.ist.psu.edu/402557.html`

[7] Johnson, B., M. Young and C. Skibo, "Inside Microsoft Visual Studio .NET," Microsoft Press, 2002.

[8] Kuhn, R., R. Chandramouli and R. Butler, *Cost effective use of formal methods in verification and validation*, in: *Foundations 02 Workshop on Verification & Validation*, 2002.

[9] Lee, B.-S. and B. Bryant, *Automated conversion from requirements documentation to an object-oriented formal specification language*, in: *SAC'02: Proceedings of the 2002 ACM symposium on Applied computing* (2002), pp. 932–936.

[10] Leitão, D., "NLForSpec: Translating Natural Language Descriptions into Formal Test Case Specifications," Master's thesis, Universidade Federal de Pernambuco (UFPE) (2006).

[11] Mesquita, W., A. Sampaio and A. Melo, *A strategy for the formal composition of frameworks*, in: *SEFM 2005, Third IEEE International Conference on Software Engineering and Formal Methods, 7-9 September 2005* (2005), pp. 404–413.

[12] Roscoe, A., *Modelling and verifying key-exchange protocols using CSP and FDR*, in: *CSFW'95: Proceedings of the The Eighth IEEE Computer Security Foundations Workshop (CSFW'95)* (1995), p. 98.

[13] Roscoe, A., C. Hoare and R. Bird, "The Theory and Practice of Concurrency," Prentice Hall PTR, 1997.

[14] Schwitter, R., *English as a formal specification language*, in: *DEXA'02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications* (2002), pp. 228–232.

[15] Schwitter, R., A. Ljungberg and D. Hood, *Ecole - a look-ahead editor for a controlled language*, in: *Controlled Translation, Proceedings of EAMT-CLAW03, Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop*, 2003, pp. 141–150.

[16] Selic, B., *Tutorial: An overview of UML 2.0*, in: *ICSE'04: Proceedings of the 26th International Conference on Software Engineering* (2004), pp. 741–742.

[17] St.Laurent, S., E. Lenz and M. McRae, "Office 2003 XML: Integrating Office with the rest of the world," O'Reilly & Associates, Inc., 2004.

[18] Wojcik, R., J. Hoard and K. Holzhauser, *The Boeing Simplified English Checker*, in: *Proceedings of the International Conference, Human Machine Interaction and Artificial Intelligence in Aeronautics and Space. Toulouse: Centre d'Etudes et de Recherches de Toulouse*, 1990, pp. 43–57.

[19] Zukowski, J., "Java 6 Platform Revealed (Revealed)," Apress, 2006.