Automated Formal Specification Generation and Refinement from Requirement Documents

Gustavo Cabral[1,2] **and Augusto Sampaio**[2]

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

[2] Centro de Informática - CIn
Universidade Federal de Pernambuco - UFPE
Caixa Postal 7851 - 50732-970 - Recife/PE - Brazil
{gflc,acas} @cin.ufpe.br

Abstract

The automatic generation of formal specifications from requirements suppresses the complexity of formal models manual creation and reveals the immediate benefits of its usage, such as the possibility to carry out refinements, and property verification, which contributes to project cost reduction and quality improvement. This paper proposes a Controlled Natural Language (CNL), a subset of English, used to write use case specifications according to a template. From these use cases a complete strategy and tools enable the generation of process algebraic formal models in the CSP notation. We define templates that represent requirements at different levels of abstraction, capturing different views of the system behavior. Moreover, a refinement notion is defined to connect the generated CSP models through an event mapping relation between abstract and concrete models. This notion is further applied to detail use case specifications and to automate its execution.

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

1. Introduction

The use of formal models, which are an abstract way to specify computer systems, is an industrial reality. Initially, the benefits regarding the use of an abstract notation, before starting the system implementation, are related to a better understanding of the problem. What has become increasingly evident is that the use of an abstract formal representation combined with techniques of model refinement can even promote the decrement of implementation time. One of the possible applications is the automatic generation of source code from formal models [24]. The testing phase has also been positively impacted by the use of models concerning test case generation [7].

As the starting point for software development, requirements need to be specially considered to produce high quality documents to serve as input to the (formal) model construction; therefore no uncertainties should remain concerning its contents. There is a variety of requirement specification methodologies, such as the one proposed in [21] that proposes a requirements elicitation process performed in six steps and an agenda for formal specification development from requirements. Nevertheless, even if requirements are appropriately captured, it is still a hard task to build models and implementations that reflect them. Usually, the transition from requirements to an analysis or design model is a manual process, and therefore error-prone.

Ideally, models should be formal, as formal methods provide the mathematical basis for achieving software correctness. Nevertheless, formal methods 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 [25] involved when developing the system formal specification. A formal approach must be cost-effective so that real projects can take advantage of formal specification benefits, 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 [36] 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 [13] also involved with natural language processing for specification validation through logic analysis. The work reported in [22] establishes a mapping between English specifications and finite state machine models. In industry, companies, such as Boeing [44], 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 [26].

Concerning the format to write requirements, use cases describe how entities (actors) cooperate by performing actions in order to achieve a particular goal. Some consensus is admitted regarding the use case structure and writing method [6]; a use case is specified as a sequence of steps forming system usage scenarios, and natural language is used to describe the actions taken in a step. This format makes use cases suitable to a wide audience.

Therefore, we build on the results achieved in [4] and propose a strategy that automatically translates use cases, written in a Controlled Natural Language (CNL), into specifications in the CSP process algebra [34]. For obvious reasons, it is not possible to allow a full natural language as a source. We define a subset of English with a fixed grammar in order to allow an automatic and mechanized translation into CSP. Because the context of this work is a research cooperation between CIn-UFPE and Motorola called CInBTCRD, the proposed CNL reflects this domain. The generated formal model is used in this project as an internal model to automatically generate test cases, both in Java (for automated ones) and in CNL (for manually executed).

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 relations between user and component views. The approach is entirely supported by tools. A *plug-in* to Microsoft Word 2003 [40] has been implemented to allow checking adherence of the use case specifications to the CNL grammar. Another tool has been developed to automate the translation of use cases written in CNL into CSP; FDR [33], a CSP refinement checker, is used to check refinement between user and component views.

The major contribution of this article, on top of the results achieved in [4], is a strategy for automated refinement of views representing models at different levels of abstraction. A refinement between two views (for instance, user and component views) requires a mapping to express a relationship between the events of these views, since, in general, each view has its own alphabet. Here we show that such a mapping can be automatically constructed from the use case templates, as well as a table relating CNL sentences provided by the use case designer. This frees the designer from operating directly with the CSP notation. As far as we are aware, this is an original approach to refinement. Furthermore, we explore several applications of our refinement strategy such as, for example, the consistence between implementation and use cases through the automatic execution of the use cases written in CNL. We also propose the definition of subviews through event decomposition. Each event from a view is further detailed into concrete events enabling its execution; it is necessary to implement the interface between the user and the system in order to automate the delivery of events to the target application. Section 6, devoted to this new approach to refinement and its applications, is entirely new. We have also further improved the presentation in [4] in several respects: the introduction to CSP, the use case templates, the formal approach to refinement, the examples and related work are all addressed in more detail.

Section 2 contains an overview of the entire solution, which includes use case templates definitions, the CNL, CSP model generation, a refinement strategy and possible applications. Section 3 contains the use case template definitions and explanations about its usage, which includes the use of the CNL; this section also contains a brief presentation of the tools implemented to support this strategy. Section 4 defines the CSP use model generation approach based on the presented use case templates and the CNL. Section 5 explores refinement between the generated CSP models, relating the user and the component views, and how refinement is mechanically checked using FDR. As explained above, Section 6 proposes a new approach to refinement, based on relating CNL sentences, as well as some applications. Finally, Section 7 summarizes our contributions, contrast the proposed solution with related work, and suggests topics for further research.

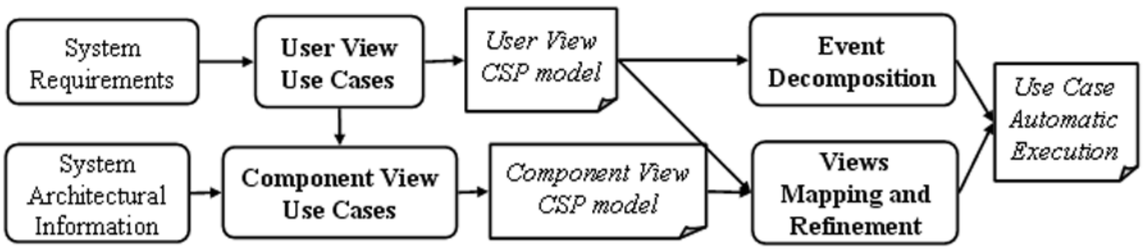

Figure 1. Proposed strategy overall process

2. Strategy Overview

In our approach, the interaction between the user and the system, or between the system components, is documented as use case specifications in a specified template. This template is structured to hold information concerning traceability with requirements, a brief description, and the way actors interact with the system. There are two use case templates: the *user view* and the *component view*.

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 proposed approach focuses on use case specifications because these are used as an input to other development phases such as design, coding, and testing. They constitute an essential part of the software development process. Therefore, we need to ensure that the visual depiction of the requirements is translated into clear and well-defined use case specifications [8].

As mentioned before, the language used to write these use cases is a Controlled Natural Language (CNL), a subset of English relevant for the specific domain. Using CNL it is possible to write imperative and declarative sentences. An imperative sentence describes actor actions and a declarative sentence depicts system characteristics, such as a GUI description or the current system state. CNL is necessary to restrict the vocabulary used to write use cases and its grammatical rules are defined by 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, 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. These events are combined with the CSP prefix operator giving 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 external choice operator, thus allowing process 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*.

Based on the generated models, a relation between user and component use cases is established by a mapping from the more abstract to the more concrete model (See Figure 1). This event mapping relation is used to prove that the component view model is a refinement of the user view model. This strategy not only enables the relation between these two views, but also allows the definition of multiple levels of abstraction within a view. The type of refinement that is explored here is called "*event decomposition*". It enables the use case designer to detail events rewriting them as more concrete ones in order to ease the understanding of its behavior.

This decomposition can be carried out as many times as necessary; the goal is to translate every step into atomic steps enabling its implementation. Consequently, this allows the automatic execution of the use cases. The automatic execution aims to verify the adherence of the use case with the implemented system, validating the consistence between the operational use case behavior and the implementation. This automation strategy is application independent; any system can be validated once the proposed event dispatching interface is implemented. For Java applications this interface can be implemented using the java.awt.Robot class or any available extension, such as that presented in [41]. The reflection mechnism, present in some programming languages, is also a possible implementation alternative.

3. Writing CNL Use Cases

Use case specifications [32] capture system behavior, possibly at different levels of abstraction. Therefore, depending on the developer's need, use cases are created for different purposes. In this section we present use case specification templates to document systems from the perspective of the user and of the system components. Both templates define execution flows that determine the interaction between the user and the system. The Controlled Natural Language (CNL), which can be seen as a processable version of English, is used to write use case steps enabling validations and transformations.

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. Table 1 presents a use case example. The following subsections explain what each use case field means and how it should be filled. The example itself is explained later.

3.1.1. Feature: Use cases are initially grouped to form a feature. Each feature contains an identification number. This grouping is convenient for organization purposes; it is not obligatory for the application of the proposed use case template. The use case itself includes a `related requirement` list, a brief `description`, and `execution flows`.

3.1.2. Related requirement(s): The related requirement list is used for traceability purposes, thus it is possible to check the use case origin. This information is also used to group use cases by requirements. When requirements change, it is possible to know which use cases might be impacted and, if it is the case, update them. Test cases related to these use cases can also be updated or regenerated (assuming an automatic approach).

3.1.3. Description: The description field gives a general idea about the use case main purpose. Since each use case is related to some requirement, any clarification about this association is also described in this field.

3.1.4. Execution Flow: A use case specifies different scenarios, depending on user inputs and actions. Hence, each execution flow represents a possible path that the user can perform. The following subsections describe each part of an execution flow.

- **Step**

  The tuple `(user action, system state, system response)` is called a *step*. Every step is identified through an identifier, an `Id`. The user action describes an operation accomplished by the user; depending on the system nature it may be as simple as pressing some button or a more complex operation, such as printing a report. The system state is a condition on the actual system configuration just before the user action is executed. Therefore, it can be a condition on the current application configuration (setup) or memory status. The system response is a description of the operation result after the user action occurs based on the current system state.

- **Flow Types**

  Execution flows are categorized as main, alternative or exception flows. The main execution flows represent the use cases *happy path*, which is a sequence of steps where everything works as expected. An alternative execution flow represents a choice situation; during the execution of a flow, such as the main flow, it may be possible to execute other actions, comprising choices. If an action from an alternative flow is executed, the system continues its execution behaving according to the new path specification. Alternative flows can also begin from a step of another alternative flow; this enables reuse of specification. The exception flows specify error scenarios caused by invalid input data or critical system states. Alternative and exception flows are strictly related to the user choices and to the *system state* conditions. The latter may cause the system to respond differently given the same user action.

- **Reference between Execution Flows**

  There are situations when a user can choose between different paths. When this happens it is necessary to define one flow for each path. Every execution flow has one or more starting points, or *initial states*, and one *final state*. The starting point is represented by the `From steps` field and the final state by the `To step` field. The `From steps` field can assume more than one value, meaning that the flow is *triggered* from different source steps. When one of the `From steps` items occurs, the first step from the specified execution flow is executed. The `To step` field references only one step; after the last step from an execution flow is performed the control passes to the step defined in the `To step` field.

  In the main flow, whenever the `From steps` field is defined as `START` it means that this use case does not depend on any other, so it can be the starting point of the system usage. Alternatively, the main flow `From steps` field may refer to other use case steps, meaning that it can be executed after a sequence of events has occurred in the corresponding use case. Yet, when the `To step` field from any execution flow is set to `END`, this flow shall terminate successfully after its last step is executed. Subsequently, the user can execute another use case that has the `From steps` field set to `START`.

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.

Exception 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

The `From steps` and the `To step` fields are essential to define the application navigation, which can be visualized as a Label Transition System [5]. These two fields also enable the reuse of existing flows when new use cases are defined; a new scenario may start from a preexistent step from some flow. Finally, loops can appear in the specification if direct or indirect circular references between flows is defined; this scenario can result in a livelock situation in the case of infinite loops.

Some considerations: The user view use case in Table 1 is an example of a mobile phone functionality. Nevertheless, this template is generic enough to permit the specification of any application, not only mobile phone ones. The user view use case holds the main characteristics of other use case definitions, such as UML use cases [35]. However, our template seems to offer more flexibility and standard. The existence of execution flows starting and ending according to other execution flows makes it possible to associate use cases in a more general way than through regular UML associations such as *extend*, *generalization*, and *include*. References enable the reuse of parts of other use cases execution flows and the possibility of defining loops, so use cases can collaborate to express more complex functionalities.

3.1.5. Use Case Example: The example in Table 1 specifies a functionality presented in most mobile phones. This use case is written using the CNL, which is detailed in Section 3.3. It specifies that messages received by a mobile phone can be moved from the inbox folder to a special folder, called `Important Messages` folder. This user view use case, in particular, includes a list of related requirements, a brief description, 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 one of the possible starting points to navigate through this system. The `To step` field is set to `END` so once the four steps from the main flow are executed the flow terminates successfully and the user can execute any use case with the `From steps` field set to `START`.

As already explained, the system state column is used to specify conditional situations. Note that this example captures only one exception flow. The normal execution of the main flow would pass through all its steps until step `4M`, after which it successfully terminates (`END`). The exception execution, which describes the situation when the `message storage is full` (system state), starts from step `1E`, just after the step `2M` is performed. In this case, given the same user action, `Select Move to Important Messages option`, depending on the system state a different system response is presented. It was not defined an alternative flow for a possible `Important Messages feature is off` state; if included, such a flow would start from step `1M`.

3.2. Component View Use Case

A component view use case specifies the system behavior based on the user interaction with system components. In this view, the system is decomposed into components that concurrently process user requests and communicate among themselves. Table 2 shows a component view use case example where the system is formed of the units `Message App`, `Message Viewer`, `Menu Controller`, `Message Storage`

`App`, `List App`, and `Display App`; each one represents different system concerns. For instance, the components `Message Storage App` and `Display App` are responsible for saving messages at the message storage and displaying notices to the user, respectively. These components design the architectural level of abstraction, refining the user view use case specified in Table 1. In other words, for each user view use case a related component view use case is defined; user view steps are decomposed into message exchange between components.

Normally, use cases describe system functionalities without revealing the internal structure of the system [35]. However, the proposed component view use cases break this convention and is actually used to detail user view use cases, which follows the regular use case idea, creating an interface between the actor and the system. The example in Table 2 is also written in the CNL; it is a refinement of the user view use case in Table 1. A formal strategy for proving this refinement is detailed in Section 5.

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. These particularities enable the definition of concurrent scenarios, which is a non-functional requirement. Thus, components can share resources and exchange messages, which is not possible in regular use case models [35].

The *execution flow* idea (main, alternative, and exception) is the same as in the user view. The system state column plays the same role as previously described in the user view (see Section 3.1.4).

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

As already mentioned, use case fields (*user action*, *system state*, *system response*, and *message*) are written in a Controlled Natural Language (CNL) with a grammar defined by knowledge bases. Using the CNL does not only make use case text clear and uniform but also allows its processing in order to generate CSP constructions.

The CNL grammar is basically a subset of English. Its sentence constructions contain domain specific verbs, terms, and modifiers. The phrases construction is centered on the verb. Domain terms and modifiers are combined to take thematic roles around the verb [10]. This strategy is detailed in [27] where it has been used to translate test case sentences into CSP constructions. The following subsections describe the knowledge bases used to store these vocables involved in the definition of the CNL.

3.3.1. Lexicon: The Lexicon stores vocables that appear in CNL sentences. Each vocable is 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. A modifier can be an adjective or an adverb that modifies a term. In the definition of each vocable, properties are associated with each one of the verbs, terms and modifiers, allowing subject-verb and noun-modifier agreement checking. The meaning of modifiers, for instance, is to qualify terms; they have no particular role besides distinguishing terms.

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 if the modifier accepts an article or not.

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.

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 [10] that contains information about the input domain verbs and its thematic roles, which can be an agent or a theme of the

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 Important 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

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`.

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.

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 first three restrict the `theme` and the `to-value` arguments, and the last one restricts only the `theme` argument; the `to-value` argument is not mandatory. Each restriction has a name; this name is used to define a CSP datatype. Finally, 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. Some Considerations

The definition of user view and component view use cases involves previous knowledge of the application requirements and architectural definitions, such as design patterns. Only after the designer is aware of these definitions and has decided which use cases are to be created, the use case writing should start.

During the creation of the two views, a relation between requirements and use cases is defined. This relation is detailed enough to point which use cases should be verified whenever requirements change. An alternative approach would be mapping requirements to steps. This would enable verifying what steps are impacted if requirements happen to change. However, this last approach revealed itself to be laborious and resulted in frequent references updates.

Analyzing the component view use case, it is easy to verify that it is actually a textual way to specify UML sequence diagrams [35]. The columns sender and receiver define actors involved in the communication and the message is the service request itself. The message order determines the sequence diagram arrangement. Besides one component sending a request to another component, the

```
<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>
```

Figure 2. Term and modifier definitions in the Lexicon

receiver component can respond this request through another message dispatch. This time the receiver acts as the sender, and vice-versa. UML sequence diagrams are also used by the Motorola development team; the automatic generation of these diagrams from a textual description represents an important benefit [9].

The adaptation of a diagrammatic language, such as UML 2.0 [38], to support features presented in the user and component views is a subject for future research. Preliminary investigation suggests that the user view use case can be modeled as a combination of the use case and the activity diagram of UML. Moreover, since the component view use case can be used to generate sequence diagrams [9], it is possible that they can actually be specified as annotated sequence diagrams. However, in spite of UML being a visual language, its practical use depends on tools and the understanding of the UML standard. Yet, the proposed approach relies on textual descriptions, possibly supported by tools as described in the next section.

3.6. Tool Support

Use case sentences must be adherent to the CNL grammar, so designers have to know the CNL grammar. There is a tool that automatically generates the CNL grammar documentation from the presented CNL knowledge bases. The CNL grammar is generated as HTML pages so it is possible to learn the CNL syntax navigating through the grammar definitions. In addition, if new domain specific terms or expressions need to be added to the CNL, it is possible to regenerate the HTML documentation. Even with this documentation, learning the CNL can be a complex task. Thus, it is recommended that the designer do not waste much time trying to figure out a way to write sentences adherent to the CNL. He should focus attention on the use case behavior.

```
<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>
...
```

Figure 3. Ontology fragment

Therefore, we have developed a tool to automatically validate the use case sentences and report all found inconsistencies. This tool is called *Use Case Validator* and it is a Microsoft Word 2003 [40] *plug-in*. It ensures use cases are written according to use case templates and the CNL syntax. MS Word 2003 is capable of structuring the use case contents through XML schema definitions. The plug-in processes the use case sentences to find inconsistencies (phrases not according to the CNL grammar). Two modules compose the plug-in. One is implemented using the .NET Platform [23] and the other is implemented in

```
<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>
```

Figure 4. Case frame example

```
<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>
```

Figure 5. Case frame and respective case frame restriction example

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 for verifying whether sentences are written according to the CNL rules. We have also implemented a Java version of the *Use Case Validator*. It reads a Microsoft Word file and reports all inconsistencies, such as sentences not following the CNL grammar.

4. CSP Specification Generation

Once use cases are created and validated using the tool mentioned in the previous section, it is possible to automatically generate CSP formal specifications.

4.1. CSP Notation

The CSP process algebra [34] is the target formalism of our strategy because it can describe complex aspects of systems, such as concurrency, in an abstract notation but still very close to implementation. It allows the description of systems in terms of processes that operate independently (parallelly), and interact (communicate) with each other, and with the environment.

The relationship between processes is described using a few process algebraic operators that allow the definition of complex process compositions. The behavior of a CSP process itself is described in terms of sequence of events, which are atomic and instantaneous operations. Through a message-passing mechanism, named channels are introduced using the `channel` keyword. These channels can transmit messages; channels can also transmit data of a specified `datatype`. As an example, we present the data *door* and two channels: *open* and *close*; the execution of the *open*!*door* event outputs the value *door* through the channel *open*. The *close* event can be similarly used to close the door. There are also two primitive processes: `STOP` and `SKIP`. `STOP` communicates nothing and stands for a canonical deadlock; `SKIP` represents successful termination.

CSP Operators: Some of the CSP operators are prefix ($a \rightarrow P$), deterministic or external choice ($P \Box Q$), and nondeterministic or internal choice ($P \sqcap Q$). The prefix operator combines an event and a process to produce a new process. The external choice operator allows the future behavior of a process to be defined as a choice between two component processes. The internal choice operator similarly 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 is selected. For example, $(a \rightarrow P) \sqcap (b \rightarrow Q)$ can behave like either $(a \rightarrow P)$ or $(b \rightarrow Q)$; it refuses to accept (engage on) a or b, and it is only obliged to communicate (transmit) an event if the environment offers both a and b. Nondeterminism is also introduced into a deterministic choice if the initial events of both sides of the choice are identical. In $(a \rightarrow a \rightarrow STOP) \Box (a \rightarrow b \rightarrow STOP)$, it is not possible to determine the system state after the occurrence of the event a.

```
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)
```

Figure 6. CSP process examples

In Figure 6, the `View_1` and the `View_2` processes are defined in *CSPm* [12], which is the machine readable version of CSP; the *CSPm* syntax enables processing by tools, such as model checkers. *CSPm* is used to define all the CSP specifications in this paper. The channels (events) `a`, `b`, and `c` are used by and constitute 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 external choice operators. For instance, after engaging on event `a`, `View_1` offers `b` and `c` to the environment. After engaging on `b` or `c` it recurses. This example is purely symbolic, but it is useful to illustrate simple CSP processes and refinement notions, which are discussed in the next section.

Other CSP operators are hiding ($P \backslash s$, where s is the set of events to be hidden), renaming ($P[[c \leftarrow d]]$), interleaving ($P \ ||| \ Q$), and the interface parallel or parallel composition ($P[| \ s \ |]Q$, where s is the set of events in which P and Q synchronize). The hiding operator provides a way to abstract processes by making some events unobservable. A trivial example of hiding is $(a \rightarrow P) \setminus \{a\}$; assuming that the event a does not appear in P, it reduces (simplifies) to P. The renaming operator replaces the occurrences of channels by other channels in a process. For instance, $P[[c \leftarrow d]]$ is a process that behaves like P except that all occurrences of channel c in P are replaced by channel d (so that c 'becomes' d).

The interleaving operator represents completely *independent* concurrent activity. On the other hand, the parallel composition operator represents concurrent activity that requires synchronization between the component processes; events in the interface set can only occur when all component processes are able to engage on that event. The parallel composition operator is also defined as $P[p \ || \ q]Q$, where p and q are sets of events accepted by the processes P and Q, respectively. In other words, p and q define the interfaces of P and Q. As an example, the process $(a \rightarrow P)[| \ \{a\} \ |](a \rightarrow Q)$ can engage on event a, and becomes the process $P[| \ \{a\} \ |]Q$, which requires that P and Q must both be able to perform event a before this event can occur. In $(a \rightarrow P)[| \ \{a, b\} \ |](b \rightarrow Q)$, we have an example of deadlock since a and b cannot be offered simultaneously.

CSP Refinement: The notion of refinement is a particularly useful concept that establishes a relation between processes (components). It captures the fact that one component satisfies at least the same conditions as another. Then we may replace a more abstract component by a more concrete one, without degrading the properties of the system. In CSP, the refinement relations are defined in three ways, depending on the adopted *semantic model*.

The *traces* refinement is based on the sequences of events which a process can perform (the *traces* of the process). A process P is a traces refinement of P ($P \sqsubseteq_t Q$), if all the possible sequences of communications that P can execute are also possible for Q; formally:

$$P \sqsubseteq_t Q \equiv traces [\![Q]\!] \subseteq traces [\![P]\!].$$

A failure is a pair (t, R), where t is a trace of the process and R is a set of events the process refuses to perform at that point. Thus, the *failures* refinement $P \sqsubseteq_f Q$ requires that the set of all failures of Q are included in the failures set of P, which means

$$P \sqsubseteq_f Q \equiv failures [\![Q]\!] \subseteq failures [\![P]\!].$$

A process is deadlocked if it can refuse to execute every event; it is commonly introduced when parallel processes do not succeed in synchronizing on the same event.

The *failures-divergences* refinement adds the concept of divergences in the failures refinement. The divergences of a process is the set of traces after which the process may livelock. This concept enhances the analysis of processes; it enables the designer to prevent the occurrence of potential situations when visible events are never performed. The failures-divergences refinement between P and Q is defines as

$$\begin{aligned} P \sqsubseteq_{fd} Q \equiv (&failures [\![Q]\!] \subseteq failures [\![P]\!]) \wedge \\ &(divergences [\![Q]\!] \subseteq divergences [\![P]\!]). \end{aligned}$$

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.

```
Read incoming message.

read.DTREA_SENDABLEITEM.(INCOMING_MESSAGE,{})

Message storage is not full.

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

Figure 7. CNL sentences and their translation to CSP events

Figure 7 presents sentences from steps `1M` and `3M` in the use case from Table 1 and their translations to CSP events. The sentence `Read incoming message` is formed of the verb `read` and its complement, `incoming message`. The verb `read` is directly mapped to the event `read` and its object is mapped to the

```
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_2M)

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_IMPORTANT_MESSAGES_OPTION,{}).(DISPLAYED_VALUE,{}) -> UC_02_3M)
  [] UC_02_1E

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_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)
```

Figure 8. Part of the generated CSP specification from the user view use case of Table 1

INCOMING_MESSAGE datatype value, which is generated from the DTREA_SENDABLEITEM case frame restriction (see Section 3.4) of the verb read. The sentence Message storage is not full contains the verb to be, conjugated as is here, 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 MESSAGE_STORAGE, FULL_STATE_VALUE, and NOT, which are used by the isstate event based on the DTISS_ITEM_STATEVALUE case frame restriction.

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. This process name is defined by the step Id combined with the use case Id, forming a unique identifier among all use case steps. Its 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. As already explained, each execution flow has From steps and To step fields that determine when the flow starts and ends. They may refer to the steps from other execution flows or to the START and END keywords.

Figure 8 shows part of 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 the START keyword (See Section 3.1.4). The process UC_02_2M is defined as a CSP external choice between the rest of the main execution flow, the process UC_02_3M, 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 their names are 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
```

Figure 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 examples of alphabet sets used in the composition.

Each component has a main process that is defined by the external choice among the component possible behaviors in each use case. Each use case gives rise to a subprocess for each component, defined by the messages exchanged between itself and other components. 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. Figure 10 shows 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. Similarly 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. Some Considerations

The user view main process, `System` (see Figure 8), is defined as the CSP external choice among the steps of use case flows that have the `From steps` field set to `START`. In contrast, the component view main process is defined as the parallel composition between system components, including the user.

Our model generation strategy is similar to [2], which generates Message Sequence Charts (MSC) from Use Case Maps [3]. However, the component view template promotes better reuse of specifications, since it is possible to reuse any sequence of steps. Like CSP, MSC offer notation to capture the concurrent aspects of the specified system. Nevertheless, CSP is a process algebra that enables the definition of channels and datatypes, along with flexible and elegant parallel operators. In addition, the CSP notation is supported by a refinement theory, refinement checkers, such as FDR [17], animators [28], and implementations, such as JCSP [43, 42] and OCCAM [16].

In the generated model, the CSP external choice operator is used to represent the user decision. The user clearly has the choice between executing a certain selected use case. However, the use of the CSP external choice operator in the alternative or exception flows seems to be a subjective issue. Because the execution of an alternative or exception flow is enabled by a combination of factors (`user action` and `system state`), the CSP internal choice operator can be considered to be used instead. For our particular application domain, however, the presence of nondeterminism in the model is irrelevant since only the traces model (See Section 4.1) is considered by Motorola during test case [31] and UML-RT sequence diagram [9] generation.

4.6. Tool Support

A tool has been implemented to mechanize the translation of the user and the component views use cases into CSP models. The tool reads user and component views use cases as Word 2003 document files, checks its content (invoking the tool presented in Section 3.6), and generates the user and the component models.

Here, the NLP module [27] is once again used to retrieve CSP events from the CNL sentences. The use model generation tool itself implements the strategy presented in this chapter; it structures the CSP events, which are effectively generated by [27], into processes to define the system formal model.

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 re-

```
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
```

Figure 10. User process exchanging messages with other components

placing abstract events with more concrete ones, formally keeping track of the relationship between the models.

In this paper we consider only two abstraction levels, user and component views. A generalization of this strategy for an arbitrary number of views is discussed in Section 6.1. Moreover, use case designers may define new use case templates and propose new ways to map events from use cases written at different levels of abstraction.

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

5.1. Refinement Mapping

We consider that the relation between user and component models is a mapping from sequences of user events to sequences of component events; to avoid nondeterministic behavior, a *one to one* relationship between sequences of events from the two models is necessary. This mapping is used by a CSP process (See Figure 11) that receives a set of pairs of sequences and yields a 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 exhibits the process that represents the mapping used in the refinement; the MAPPING process receives the mapping between the two views and through the TRIGGER function defines an indexed external choice among the processes generated by the makeProcess auxiliary function. This last function receives one sequence that is initiated with the events from the abstract model followed by events from the concrete model and recursively uses the prefix operator to create a process terminated with the SKIP process.

```
MAPPING(map) = TRIGGER(map); MAPPING(map)

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

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

Figure 11. Mapping process

The process that represents the mapping is composed, through an alphabetized parallel composition, with the abstract model. This composition contains events from both views. Once the events from the abstract model are hidden, it produces a process that must be 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 concrete models. 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 the `MAPPING` process. 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(map)
View_1_mapped = View_1_with_mapping
                 \events_view_1

View_1_mapped [FD= View_2
```

Figure 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 [30]. Here we focus on relating events from different models for refinement purpose, while the framework composition strategy aims to accomplish communication between frameworks possibly with different alphabets.

5.2. Component View as a Refinement of the User View

The same mapping strategy presented in the previous section is 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.

Figure 15 presents part of the `mapping` between the user and the component view events. The event related to step `1M` from the user view is mapped to the ones related to steps `1M` and `2M` from the component view, and the user view event from step `2M` is mapped to the component view events of steps `3M`, `4M`, and `5M`, establishing a relation between user and component views (Figure 13).

As explained, the component view refines the user view through events mapping. In some cases, it is also possible to retrieve the user view from the component view, provided an inverse mapping from the component to the user view.

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

User_View_mapped = User_View_with_mapping
                   \events_user_view

User_View_mapped [FD= Component_View
```

Figure 13. Mapping process specification based on the map

5.3. Tool Support

The refinement relation discussed here can be mechanically checked using FDR [33], 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 14, 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 13. The results established that both refinements hold, as expected.

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

Figure 14. Assert commands verified by the 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.

The generation of the mapping that relates the user and the component views can be automated since 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 can be used to assist the event mapping task suggested in Section 6.

6. A Strategy for Automated Refinement and its Applications

The event mapping strategy presented in Section 5 enables the relation, or tracking, between events expected by the system in the user view and the behavior defined in the system design, which is represented here as the component view. While establishing a formal mapping is essential to ensure consistency between views, this task is usually considered as a barrier to the practical application of formal refinement.

```
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,{}) > ), ... }
```

Figure 15. Mapping between abstract and concrete views

Concerning the proposed approach, in particular, it requires the direct manipulation of the CSP process algebra. Nevertheless, use case designers are not usually familiar or willing to work with formal notations. Our approach to automated refinement is to construct the relevant mappings from information provided by the use case designer at the use case level; this information is described in terms of CNL phrases. The actual use of the CSP notation is hidden from the final user.

Considering the relationship between the user and the component views, formalized in the previous section, each step from the user view is selected and associated with one or more steps from the component view. This selection and mapping process is entirely accomplished without the manipulation of CSP events; only CNL phrases are handled by the use case designer.

Step Id	User Action	System State	System Response
1M	Read incoming message.		Message content is displayed.

⇓

Step Id	Sender	Message	System State	Receiver
1M	User	Read incoming message.		Message App
2M	Message App	Open incoming message.		Message Viewer

Table 3. Example of event mapping from user to component view

As an example, Table 3 shows the step `1M` from the user view use case and its respective refinement, which are the steps `1M` and `2M` from the component view use case. Thus, once the user-component mapping is defined by the use case designer, manipulating CNL phrases only, we can generate the respective CSP events from these phrases and use them to produce the mapping definition that is necessary for the refinement strategy presented in Section 5. The rest of this section presents possible applications for the mapping between views, which can be automatically generated with the use case designer assistance and tool support.

6.1. Event Decomposition

An obvious application of event decomposition is relating sequences of events from different views as already discussed. However, this idea can be generalized.

Taking the user view as an example to apply the event decomposition strategy, note that it is possible to express the user behavior into several levels of abstraction. One simple phrase could actually define a complex action and therefore need to be decomposed in several simpler actions. The classification of an action as complex or simple is related to the possibility of breaking it down in several sub-actions. This process would occur in an incremental way until the initial complex action is mapped to a sequence of atomic actions (or events) that does not need to be further detailed. At this point, it is necessary to determine the atomic events according to the interface between the user and the system. In other words, these are the concrete events expected by the application.

Figure 16 shows how a sequence of events in a certain `Level 1` of abstraction has its events broken down into more concrete ones from `Level 2`. Notice that the `Event A` is initially decomposed into the sequence `Event A.1, Event A.2, and Event A.3`. Observing the abstraction `Level 3`, we can see that not all events from `Level 2` need to be further decomposed. Only the event `Event A.2` needs to be detailed as `Event A.2.1` and `Event A.2.2`. In general, an arbitrary hierarchy level is allowed; a sequence

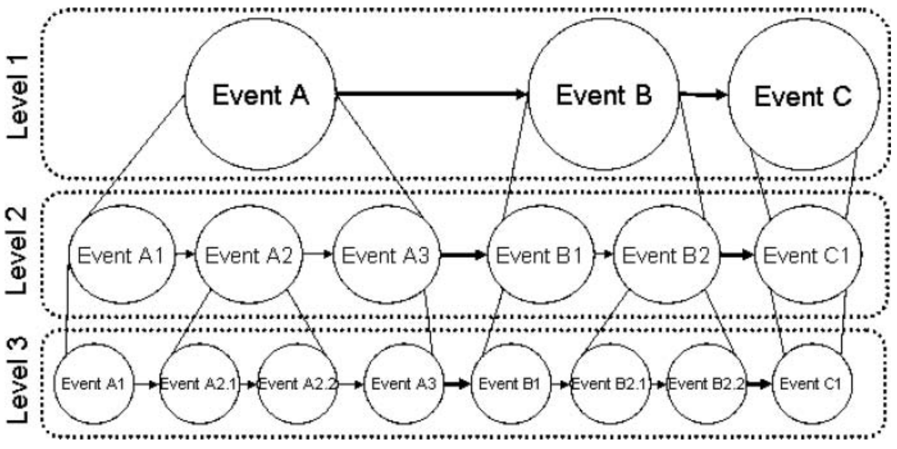

Figure 16. Illustration of the event decomposition strategy

of events from a more abstract view is mapped to a sequence of events from a more concrete view, just as in the refinement strategy presented in Section 5.

6.1.1. CNL Atomic Events: The definition of atomic events requires a close analysis of the system interface; all possible events accepted by the application should be listed. Once this set of atomic events is defined, it is necessary to add them, as verb definitions, to the CNL knowledge bases so they can be used by the CNL parser. The `press` operation, for instance, defines an atomic event present in mobile phone applications; keys are pressed by the user to interact with the phone. Figure 17 defines the `press` verb, case frame, and frame restriction. The `press` verb does not require a subject or an agent, it requires a theme that is the key to be pressed.

The `press` verb accepts verbal complements that are members of the `DTPRESS_PRESSABLEKEY` class from the Ontology. This class is composed by nouns representing the possible keys that can be pressed by the phone user. Thus, CNL user actions can now be written as atomic events (concrete actions).

```
<verb>
   <name>press</name>
   <third-person>presses</third-person>
   <gerund>pressing</gerund>
   <past>pressed</past>
   <participle>pressed</participle>
</verb>

<frame>
  <description>Press a key. Ex.:
  Press the menu key</description>
  <name>pressKey</name>
  <verblist>
     <verb>press</verb>
  </verblist>
  <roles>
     <role mandatory="False">agent</role>
     <role mandatory="True">theme</role>
  </roles>
</frame>

<frame>
  <name>PressKey</name>
  <restrictions>
    <restriction name="DTPRESS_PRESSABLEKEY">
     <class role="theme">pressable_item</class>
    </restriction>
  </restrictions>
</frame>
```

Figure 17. Definition of `press` verb, case frame and frame restriction

6.1.2. Decomposition Strategy: As presented in Section 3.6, the use cases are written using Microsoft Word and the CNL is validated by a *plug-in*. The use case is structured by an XML schema that annotates the document content enabling the *plug-in* to access each part of the use case. The mapping presented in the beginning of this section (See Table 3) can be similarly defined using XML annotations over the use cases. This strategy is used for arbitrary event mapping between views and, particularly, for event decomposition, mapping each user abstract action to a sequence of atomic concrete actions.

In Table 4, we have examples of user actions and the respective atomic action. Since user definitions of events can be quite abstract, the decomposition of such events may occur in several stages. The action `Select 'Move to Important Messages' option`, for instance, is decomposed into the atomic action `Press down arrow key`, which happens twice.

6.2. Use Case Execution

The decomposition of events, all the way into atomic user actions, not only details the use case definition but

User Action	Atomic Actions
Read incoming message.	Press center key.
Confirm memory information dialog.	Press left soft key.
Select "Move to Important Messages" option.	Press down arrow key. Press down arrow key.

Table 4. Decomposition of user events into atomic events.

may enable its execution. Once the set of atomic events is defined for a particular application domain, it is possible to simulate the actual execution of its events. These atomic events are mapped to an interface that dispatches them to the system, simulating the user behavior.

```
channel press : DTPress

datatype DTPress = DTPRESS_PRESSABLEKEY.
                   (KeyValue,Set(Modifier))

datatype KeyValue = MENU_KEY | CENTER_KEY |
  RIGHT_SOFT_KEY | LEFT_SOFT_KEY | ...

                      ⇓

public interface AtomicEvents {
  public void press(KeyValue key);
}

public enum KeyValue {
  menu, center, rightSoft, leftSoft ...
}
```

Table 5. `Press` channel and associated datatype

Table 5 illustrates part of the CSP specification that defines the `press` channel and its associated datatypes; this definition is automatically derived from the CNL knowledge bases. Notice that the `DTPress` datatype holds the `KeyValue` information and a possible set of `Modifiers`. Here, each CSP channel is mapped to a method in the `AtomicEvents` Java Interface and the datatypes are implemented as Java Enumeration. This Java code can be also automatically generated from the CNL knowledge bases.

6.2.1. Mobile Phone Automation: At this point, the automatic execution of an application depends only on the implementation of the presented interface. In the case of a Motorola mobile phone, there is an API that allows the access of the phone's current state and event dispatching. Such an API enables the implementation of the presented interface. Thus, once a user view use case is written and each step is decomposed into atomic events, it is possible to execute it, therefore verifying its consistence with the implementation. This procedure is seen as a preliminary development of the necessary infra-structure to enable use case or test case automatic execution.

In the case of test case execution, its generation from the user view CSP model (See Section 4) is achieved applying strategies such as the one defined in [31]. This strategy allows the definition of test purposes (as CSP processes) that filter the generated model and yields traces according to specified guidelines.

6.2.2. Desktop Application Automation: Since the event decomposition strategy manipulates only CNL phrases in the use case specifications, it is extensible for different application domains, such as Desktop or Web applications. It defines a Platform Independent Model (PIM) [24]. It is a matter of updating the CNL knowledge bases with domain terms and define the set of atomic events (application interface) in order to enable the execution of use cases (or test cases) in the new application domain. Thus, once the interface is implemented, it is possible to execute new use cases without extra effort.

This strategy was initially applied for Java Desktop applications since the Java platform implements an interface that provides access to the application's running objects, such as Graphical User Interface (GUI) components. The java.awt.Robot class intercepts a running Java application and allows access to variable values and event dispatching, enabling the implementation of user actions.

In the Desktop application domain, we have defined a new set of atomic events formed by the `click` and the `type` events. Table 6 contains part of these event definitions. The `click` event denotes the mouse click action performed on a specified item, such as a button or menu identified by a name (`ClickValue`). The `type` action represents the interaction of the user through the keyboard, enabling data input.

The java.awt.Robot class can be directly used to access the running system. Nevertheless, more robust frameworks have been implemented to facilitate the access of the application GUI components. Jemmy [41] is a framework that provides a high level API to capture application state and perform actions. It is used in an experimental implementation of the Java interface defined in Table 6 to validate the presented technique.

6.3. Some Considerations

The event decomposition strategy presented in subsection 6.1 is based on the formalism explained in early sections and enables the use of formal methods for the use case execution purposes. The strategy itself requires assistance from the use case designer, such as interface implementation, but the results seem promising.

The decomposition strategy allows substantial reuse

```
channel click : DTClick
datatype DTClick = DTCLICK_ITEM.
                   (ClickValue,Set(Modifier)).
                   (ClickItem,Set(Modifier)).

datatype ClickValue = OK | CANCEL |
    EDIT | OPTION | BACK ...

datatype ClickItem = MENU | SUBMENU |
    BUTTON | FIELD | IMAGE ...

channel type : DTType
datatype DTType = DTTYPE.
                  (TypeValue,Set(Modifier))

datatype TypeValue = JOHN | MARY |
    RECIFE | CIN | ...
```

$$\Downarrow$$

```
public interface DesktopAtomicEvents {
  public void click(ClickValue value,
                    ClickItem item);
  public void type(TypeValue value);
}

public enum ClickValue {
  ok, cancel, edit, back, ...
}
public enum ClickItem {
  menu, subMenu, button, field, ...
}
public enum TypeValue {
  john, mary, recife, cin, ...
}
```

Table 6. `Click` and `type` channel and associated datatypes

of user action definitions; once a user action is mapped to atomic events, this mapping is reused in the automation of other use cases without the necessity of decomposing the same user action again. Similarly, the atomic event interface only needs to be implemented once for a particular application domain; new use cases are immediately automated after their events are decomposed.

Initially, the definition of atomic events can be complex; it is difficult to determine a generic interface for a wide range of applications. It is still necessary to investigate the variety of possible ways users can interact with systems. The rise of new types of GUI components, for example, forces the definition of new atomic events and their respective implementations.

Nevertheless, the use of such a strategy can be further extended enabling the definition of scripts, as in a script language, in order to automate tasks frequently executed by the user. The execution of test cases still requires further analysis since test cases involve not only execution of user action but verification of the system responses.

7. Final Considerations

The usage of formal methods with the purpose of documenting system and consequently enabling the generation of test cases and other artifacts is being unusually explored in this paper. The use of formal methods is commonly related to the definition of a specification that can be refined and eventually mapped to code, thus guaranteeing the quality of the implementation; such usage is not the main goal here.

The cooperation with a company, such as Motorola Inc., brought a practical appeal to the accomplishment of this research. The proposed strategy focuses on generating formal specifications 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. In addition, the validation of the use cases behavior reinforces the adequacy of the specified scenarios.

The use of natural or restricted languages to write requirements is approached by various works [18, 13, 14, 15] that generate first-order logic models. However, this strategy seems to be more suitable for requirements consistence verification. Using only a logical notation to specify system architecture and design seems inappropriate; the gap between logical propositions and structured design is wide. Nevertheless, the use of CNL to ensure requirements consistence and specify systems seems promising. In addition to that, CNL editors [13, 37] are a viable solution to enable the use of CNL, minimizing any negative impact.

Processing use case specifications to generate formal models is brightly addressed in [29], however the proposed CNL presents a clear definition of grammar that is simple to learn, use and extend. We also define structured use case specification templates. The implementation of the CNL editor is a possible future work.

In [39] an approach is defined to generate CSP models from policies. However, the definition of policies to specify a system seems confusing. The use of the proposed use case specification templates seems to favor a better understanding of the system behavior.

An approach similar to ours is presented in [3] where a notation called Use Case Maps (UCMs) is introduced to allow the design of scenarios at a more abstract level in terms of sequences of responsibilities over a set of components, just as in the component view. Still, UCMs do not model explicit inter-component communication, but it can be translated to Message Sequence Chart (MSC) [20] specifications [2]; MSC is also supported by model checkers [1], allowing property verification. This combination of strategies needs to be further investigated.

Apart from the fact that we use a process algebra as a 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: the use case specification creation, the refinement checking, and the use case execution. These tools are essential to the introduction of formal methods in real projects, as in the Motorola environment.

The major 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 [5]. There is also complementary work in the CInBTCRD research project that uses the proposed component view model to generate UML diagrams; in [9] a set of laws is proposed to map CSP specifications into UML-RT diagrams, which is now part of version 2.0 of UML.

Besides automatic generating test cases from formal models, there are means to animate the formal model [11] and trigger the execution of commands that shall execute operations at the real application. These operations would result in system responses, which can be verified using the system response definition from the user view model. In other words, the formal specification can be executed through an animator and the real application would concurrently receive concrete stimuli from the environment.

The proposed model refinement strategy, through events mapping, and the use case execution approach can also be used as an important step towards automating test case execution. The execution of user actions based on atomic events associated with automatization of test case verification enables 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.

Acknowledgments

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] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. In *CONCUR'99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 114–129. Springer, 1999.

[2] F. Bordeleau. *A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical Finite State Machines*. PhD thesis, Carleton University, 1999.

[3] R. Buhr. Use Case Maps as architectural entities for complex systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, 1998.

[4] Gustavo Cabral and Augusto Sampaio. Formal specification generation from requirement documents. In *Brazilian Symposium on Formal Methods (SBMF)*, pages 217–232, 2006.

[5] Emanuela Cartaxo. 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.

[6] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[7] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz. Model-based testing in practice. In *ICSE'99: Proceedings of the 21st international conference on Software engineering*, pages 285–294. IEEE Computer Society Press, 1999.

[8] Brian Dobing and Jeffrey Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, 2006.

[9] Patricia Ferreira, Augusto Sampaio, and Alexandre Mota. Viewing CSP specifications with UML-RT diagrams. In *Brazilian Symposium on Formal Methods (SBMF)*, pages 73–88, 2006.

[10] C.J. Fillmore. 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.

[11] Angela Freitas and Ana Cavalcanti. Automatic translation from Circus to Java. In *Lecture Notes in Computer Science : FM'2006: Formal Methods*, volume 4085, pages 115–130. Springer, 2006.

[12] Leonardo Freitas, Ana Cavalcanti, and Hermano Moura. Animating CSPm using Action Semantics. In *Proceedings of IV Workshop em Métodos Formais*, pages 58–69. Sociedade Brasileira de Computacão (SBC), 2001.

[13] N. Fuchs, 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*, pages 1–20. Springer, 1990.

[14] N. Fuchs, U. Schwertel, and S. Torge. Controlled natural language can replace first-order logic. In *ASE'99: Proceedings of the 14th IEEE international conference on Automated software engineering*, page 295. IEEE Computer Society, 1999.

[15] N. Fuchs and R. Schwitter. Specifying logic programs in controlled natural language. Technical Report ifi-95.17, University of Zurich, 1995.

[16] John Galletly. *Occam-2*. University College London Press, 1996.

[17] P. Gardiner. *Failures-Divergence Refinement, FDR2 User Manual and Tutorial*. Formal Systems Ltd., 1997.

[18] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transactions on Software Engineering Methodology*, 14(3):277–330, 2005.

[19] Mark Grand. *Java language reference*. O'Reilly & Associates, Inc., 1997.

[20] David Harel and P. Thiagarajan. Message sequence charts. In *UML for real: design of embedded real-time systems*, pages 77–105. Kluwer Academic Publishers, 2003.

[21] Maritta Heisel and Jeanine Souquières. A method for requirements elicitation and formal specification. In *ER'99: Proceedings of the 18th International Conference on Conceptual Modeling*, pages 309–324. Springer, 1999.

[22] Alexander Holt. Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, 24:253–257, 1999.

[23] Brian Johnson, Marc Young, and Craig Skibo. *Inside Microsoft Visual Studio .NET*. Microsoft Press, 2002.

[24] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[25] Richard Kuhn, Ramaswamy Chandramouli, and Ricky Butler. Cost effective use of formal methods in verification and validation. In *Foundations'02 Workshop on Verification & Validation*, 2002.

[26] Beum-Seuk Lee and Barrett 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*, pages 932–936. ACM Press, 2002.

[27] Daniel Leitão. NLForSpec: Translating natural language descriptions into formal test case specifications. Master's thesis, Universidade Federal de Pernambuco (UFPE), 2006.

[28] Formal Systems (Europe) Ltd. *PROBE Users Manual version 1.25*. Formal Systems (Europe) Ltd, 1998.

[29] Vladimir Mencl. Deriving behavior specifications from textual use cases. In *WITSE'04 - Workshop on Intelligent Technologies for Software Engineering*, pages 331–341, 2004.

[30] Walter Mesquita, Augusto Sampaio, and Ana Melo. A strategy for the formal composition of frameworks. In *SEFM'2005, Third IEEE International Conference on Software Engineering and Formal Methods*, pages 404–413. IEEE Computer Society, 2005.

[31] Sidney Nogueira. Geracão automática de casos de teste CSP dirigida por propósitos. Master's thesis, Universidade Federal de Pernambuco (UFPE), 2006.

[32] Colette Rolland and Camille Achour. Guiding the construction of textual use case specifications. *Data Knowl. Eng.*, 25(1-2):125–160, 1998.

[33] A.W. Roscoe. Modeling and verifying key-exchange protocols using CSP and FDR. In *CSFW'95: Proceedings of the The 8th IEEE Computer Security Foundations Workshop*, page 98. IEEE Computer Society, 1995.

[34] A.W. Roscoe, C.A.R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[35] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.

[36] R. Schwitter, A. Ljungberg, and D. Hood. ECOLE - a look-ahead editor for a controlled language. In *EAMT-CLAW'03 - Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop*, pages 141–150, 2003.

[37] R. Schwitter, A. Ljungberg, and D. Hood. ECOLE - a look-ahead editor for a controlled language, in: Controlled translation. In *EAMT-CLAW'03 - Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop*, 2003.

[38] Bran Selic. Tutorial: An overview of UML 2.0. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 741–742. IEEE Computer Society, 2004.

[39] R. Sterritt, M. Hinchey, J. Rash, W. Truszkowski, C. Rouff, and D. Gracanin. Towards formal specification and generation of autonomic policies. In *EUC Workshops*, pages 1245–1254, 2005.

[40] Simon St.Laurent, Evan Lenz, and Mary McRae. *Office 2003 XML: Integrating Office with the rest of the world*. O'Reilly & Associates, Inc., 2004.

[41] Yanhong Sun and Edward Jones. Specification-driven automated testing of GUI-based Java programs. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 140–145. ACM Press, 2004.

[42] Peter Welch. Process Oriented Design for Java: Concurrency for All. In *Computational Science - ICCS'2002*, volume 2330, pages 687–687. Springer, 2002.

[43] Peter Welch, Jo Aldous, and Jon Foster. CSP networking for Java (JCSP.net). In *ICCS'02: Proceedings of the International Conference on Computational Science-Part II*, pages 695–708. Springer, 2002.

[44] R. Wojcik, 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*, pages 43–57, 1990.