Unifying Models of Test Cases and Requirements

Clélio Feitosa[1], Glaucia Peres[1] and Alexandre Mota[1]

[1] Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851 - 50732-970 – Recife – PE – Brazil
{cfs, gbp, acm}@cin.ufpe.br

Abstract. In industry, due to market pressures, it is common that the system requirements are out of date or incomplete for certain parts of the system. Nevertheless, we can always find up to date test cases which implicitly complements the related requirements. Therefore, instead of simply using test cases to detect software failures, in this paper we present an approach to update requirements using test cases. To accomplish this, we first assume that both requirements and test cases are formally documented; we reuse previous works that provide such models automatically as CSP formal specifications. Thus, we formally define a merge operation using the operational semantics of CSP. Finally, we use part of a real case study to experience the proposed approach.

Keywords: formal methods and models, CSP, CNL, modelling and unifying models.

1 Introduction

It is well-known from Software Engineering that a good requirements document is a required starting point to quality software development [1]. From requirements, one originates the architecture of the system, considers design decisions, builds the implementation, and specifies test cases among other tasks. For instance, inside the requirements document we can find the requirements specification (or model) which, in general, only considers the functional aspects of the product.

Unfortunately, in real (or industrial) software development, due to associated market pressures to release new product versions as soon as possible, some requirements are unavailable or out of date. But interestingly, as the product must be released, the implementations and test cases stay up to date. This situation is not so difficult to explain. Initially, we have an up to date requirements specification. Later, during development, it is sometime forgotten (the team has no time to update it), giving priority to what the market wants: software running (code) and quality (tests).

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

In this paper we propose an approach to automatically update the (out of date) requirements specification using (up to date) test cases. To accomplish this, we reuse a controlled natural language---CNL (Section 2), developed in the context of the

CIn/BTC research project, to describe both requirements and test cases. Thus, we translate both documents into the process algebra CSP (Section 3), used as our formal basis, where we reuse the translation CNL-CSP and present one for test cases to CSP. Finally, we propose a merge operator (Section 5) using the operational semantics of CSP, which takes two CSP models and returns the unifying one. In Section 6 we show a tool support for our proposal and use it in a simple case study. Finally, we present our conclusions and future works in Section 7.

2 The Controlled Natural Language

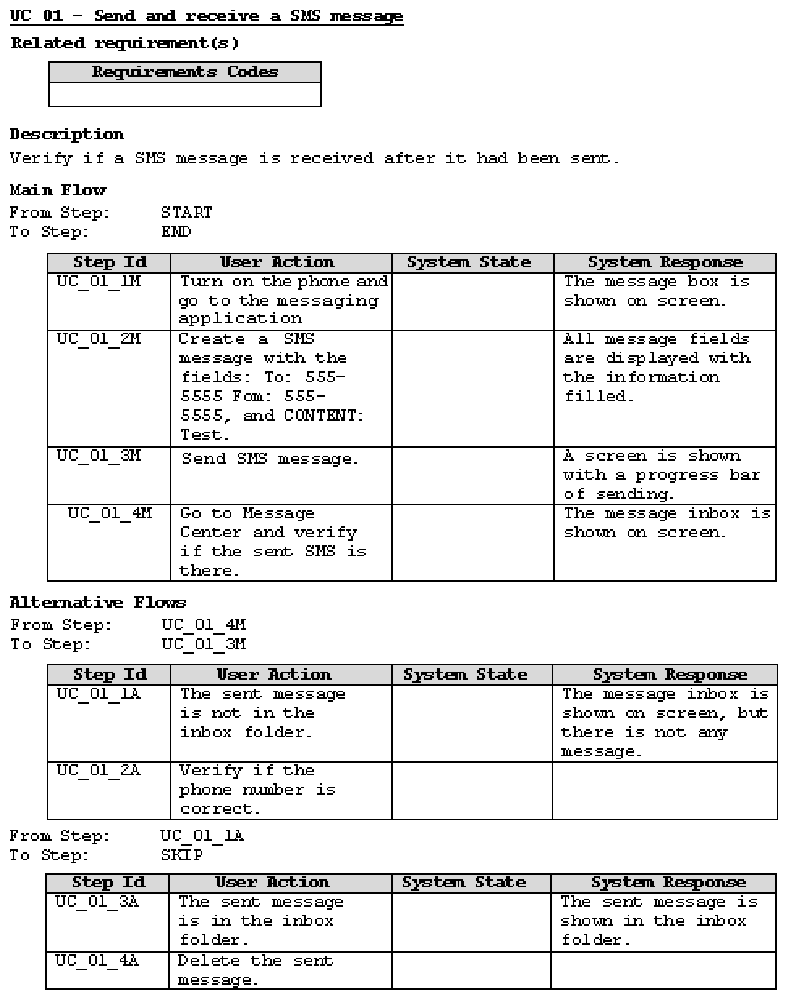
Controlled Natural Language (CNL) is a processable version of English. The CNL grammar is a subset of the English grammar. Its sentences contain domain specific verbs, terms, and modifiers. The phrases are centered on the verb. Domain terms and modifiers are combined in order to take thematic roles around the verb [2]. This strategy is detailed in [3] where it has been used to translate test cases sentences into CSP constructions. Fig. 1 presents a simplified view of the syntax of the CNL. Thus a requirements document is seen as a set of CNL sentences.

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

Fig. 1. BNF for the Controlled Natural Language

In Fig. 2, we observe an example of a use case where the fields `User Action` and `System Response` are written in the CNL. The example is a template of a use case document proposed by [4] and it is used to represent the use cases involved in our approach. There are two types of flows: the `Main Flow` and the `Alternative Flows`. For each flow we have the step description (represented by the field `User Action`), the `System State` that complements the steps and the `System Response`. Each alternative flow is related with some step of the `Main Flow` or with some step of another `Alternative Flow`, by the field `From Step`. The field `Step Id` is used to make each step unique in the use case. In our example, the step `UC_01_1M` represents the first step of the use case. The field `To Step` indicates which step follows the last step of its corresponding flow. For instance, the `To Step` of the last alternative flow in Fig. 2, filled with `SKIP`, says that after the step `UC_01_4A`, the execution of the flow is over. In this example the

`Requirements Codes` field is empty, meaning that the use case `UC_01` is not related to other use cases.



UC 01 – Send and receive a SMS message

Related requirement(s)

Requirements Codes

Description

Verify if a SMS message is received after it had been sent.

Main Flow

From Step: START
To Step: END

Step Id	User Action	System State	System Response
UC_01_1M	Turn on the phone and go to the messaging application		The message box is shown on screen.
UC_01_2M	Create a SMS message with the fields: To: 555-5555 Fom: 555-5555, and CONTENT: Test.		All message fields are displayed with the information filled.
UC_01_3M	Send SMS message.		A screen is shown with a progress bar of sending.
UC_01_4M	Go to Message Center and verify if the sent SMS is there.		The message inbox is shown on screen.

Alternative Flows

From Step: UC_01_4M
To Step: UC_01_3M

Step Id	User Action	System State	System Response
UC_01_1A	The sent message is not in the inbox folder.		The message inbox is shown on screen, but there is not any message.
UC_01_2A	Verify if the phone number is correct.		

From Step: UC_01_1A
To Step: SKIP

Step Id	User Action	System State	System Response
UC_01_3A	The sent message is in the inbox folder.		The sent message is shown in the inbox folder.
UC_01_4A	Delete the sent message.		

Fig. 2. Example of a use case written in the CNL

Fig. 3 illustrates two examples of CNL sentences. The first one, `Delete the sent message`, is an imperative sentence that has the verb `Delete` and the term `sent message`. In this case, thematic roles are defined to the verb `Delete` in order to determine how the verb `to delete` is associated with terms. Thus, the positions of the terms in the sentence are also defined by the thematic roles. The same happens to the second sentence, `All message fields are displayed`, where thematic roles are also defined to the verb `to be` in order to validate the construction `<term> <verb> <modifier>`.

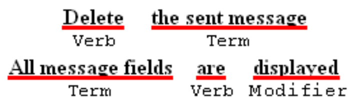


Fig. 3. CNL sentences components

3 An Outline of CSP

The process algebra CSP [5] is a formal specification language specially designed to describe the behaviour of concurrent systems. However, CSP can also be seen as a mathematical theory to study processes, which interact with each other and their environments, using communications. CSP is based on a synchronous communication model. In practice, we use a machine-readable version of CSP called CSP$_M$ [6]. It basically adds a functional language to CSP to deal with data aspects of a system as well as has an ASCII representation for all elements of CSP. CSP$_M$ is used in the CSP model-checker FDR [7] and animator ProBE [8]. The former allows us to check properties, such as deadlock and refinements automatically, whereas the latter allows us to investigate the behaviour of a CSP model interactively.

The fundamental idea of CSP is a communication based on events; events are instantaneous (atomic) actions that a system can perform. Events are assumed to be drawn from a set Σ; the *alphabet* of a specification. Thus, the set Σ contains all possible communications a CSP specification can perform. In a specification, events are introduced by `channel` declarations; channels can be typed or untyped. Untyped channels (`turnOn`) are indeed events (or simply synchronization points), whereas typed channels (`send` and `receive`) are collection of events (also seen as vehicles to communicate data of the respective type). Types are built-in (`Int` for integers) or user-defined (`datatype` denotes enumerations and `nametype` abbreviations). The following fragment of CSP specification illustrates these elements.

```
datatype MESSAGES = SMS | EMS | MMS
nametype PHONE = Int
channel turnOn
channel send, receive: MESSAGES.PHONE
```

A typed channel `send` can communicate any data from the composed type `MESSAGES.PHONE`. Thus, an event from this collection is an element whose name is `send`, followed by `(.)` a value formed from the enumerated type `MESSAGES` (or an `SMS`, `EMS` or `MMS` message), followed by `(.)` a phone, specified by the type `PHONE` (or an integer). For instance, `send.SMS.1` defines the action of sending an SMS message to phone `1`. Typed events can be used to input or output data. To describe an input using the channel `receive` we write `receive?msg?phone`. This pattern introduces two variables msg and phone where their values are only determined after a communication has occurred successfully. In order to input or output data, synchronize communications and describe behaviours, CSP uses processes. A process is a behavioural unit. A BNF of a CSP process is.

```
P ::= STOP
| SKIP
| a -> P
| P |~| P
| P [] P
| P [|X|]P
| Q
```

In CSP, two basic processes deserve special attention: The first is the process STOP, which represents a broken machine (that is, a machine that is unable to communicate anything), and the second is the process SKIP used to denote a successfully terminated process. SKIP differs from STOP by simply performing a special event √ (tick) before terminating and behaving like STOP. The simplest behaviour apart from STOP and SKIP is given by the prefix (->) operator. The process e -> P offers the event e to its environment[1] and waits indefinitely for its occurrence. Once the event e occurs, the process e -> P behaves like P.

Alternative behaviour is characterized by two kinds of choice operators: The internal or non-deterministic choice (|~|) and the external or deterministic choice ([]). The process P |~| Q behaves like P or Q independent of the environment. So, the process P |~| Q passes to behave like P or Q and only after that the environment can interact with it. On the other hand, the process P [] Q can behave like P or Q but the choice is decided by the environment. If the environment is only able to interact with P, then P is chosen, otherwise Q is chosen.

Sometimes it is necessary to model independent machines that can eventually communicate to each other. Thus, the parallel ([|X|]) operator is used. The process P [|X|] Q can behave like P′ [|X|] Q, in case P can evolve to P′, and like P [|X|] Q′, if Q can evolve to Q′. If both P and Q can evolve but the environment cannot distinguish which of them has evolved, then a non-deterministic behaviour occurs. These situations occur when P and Q cannot interact with each other. If they interact then the previous process behaves like P′ [|X|] Q′, that is, both processes must evolve simultaneously (synchronously).

A CSP process is defined by an equation, where the left-hand side has the name of the process with eventual parameters and the right-hand side has the process body, given by a composition of the previous operators. The following equation defines the process TEST_CASE_1 as a series of prefixes terminating successfully.

```
TEST_CASE_1 =
   setup -> goTo.MESSAGE_EDITOR.1 -> create.SMS.1 ->
   test -> send.SMS.1 -> receive?mesg.1 -> open.mesg ->
   cleanup -> delete.mesg -> goTo.IDLE -> SKIP
```

[1] A CSP environment is anything which can interact with a process.

To capture non-terminating behaviour, we simply use recursive processes. The last element of our BNF refers to a recursive process call (`Q`). This occurs when in the right-hand side of an equation, `P = ... Q`, the name of a process corresponds to the left-hand side of some (other or the same) equation (`Q = ...`). In this case, the behaviour is simply transferred to the right-hand side of the called process. Therefore, recursive behaviour allows infinite system modelling.

To state formally the meaning of a CSP process various semantics were proposed [5]. At least three distinct ways are proposed to gain a rigorous mathematical understanding of what a CSP program means. They are the operational, denotational, and algebraic semantics. In this paper, we consider the CSP operational semantics to propose the merge operator, described in Section 5.

4 CSP Models of Test Cases and Requirements

To unify documents automatically, an obvious starting point is that the artefacts (test cases and requirements) being written in a formal language. In this paper we use the process algebra CSP as this medium. Prior to present how the unification is performed we must present how our artefacts are generated in terms of CSP, which in general is a difficult task itself. In this work we show that this task is relatively easy to satisfy once we consider a specific domain, mobile phone applications.

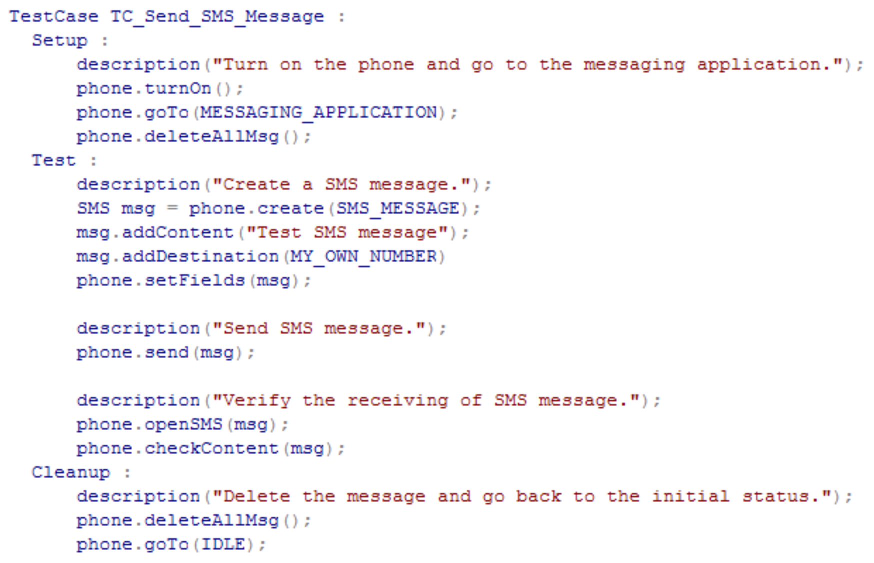
Section 4.1 gives an overview of how the CSP model is generated from standard test cases, and Section 4.2 for models originated from system requirements.

4.1 From Test Cases to CSP Model

The generation of a CSP model from a set of test cases is performed automatically. First, each test case is translated to CSP and thus the various intermediate CSP representations (CSP test cases) are grouped using a CSP function. In what follows, we present a brief idea about this translation task.

In general, in industry, we have two kinds of test cases: one is informal and described in some natural language, and another is precise having a programming language as basis. Informal test cases are a central artefact for human beings but are not so useful for automatic testing. In this latter case, precise test cases are employed. A test script, or test case script, is a kind of precise test case defined by a series of steps based on some programming language. To ease the understanding of this translation process, we consider an instance of a test script (see Fig. 4) and its generated formal test case. The test script of Fig. 4 is related to the use case described in Fig. 2 and it is composed by three main parts: setup, test, and cleanup. Each part has a set of actions, where each action is described in a CNL (**description**), corresponding to its related use case's step. Each description represents a goal of a specific step (set of actions) of the test. Associated to each action, we have a sequence of script commands. The translation consists in taking each goal (**description** field) of the test described in CNL and generating a corresponding CSP event. In general terms, the translation CNL-CSP follows a natural language processing [3], where the verb of the sentence (main element) is converted to a typed CSP event or channel. The verbal complements mean the composed data of the typed channel, which are supposed to communicate. Beyond the generated CSP events, such translation is also responsible for generating all CSP header definitions, such as type, event and channel definitions, and so on. The detailed description of this translation task is out of the present work scope. For further details consider the work presented in [3].

A CSP test case is represented as a CSP trace, where each event corresponds to an action of the test case. In what follows, we show an instance of a CSP test case generated from the test script shown in Fig. 4. That is, we present the trace `TC_Send_SMS_Message` in terms of CSP. Such a representation is similar to any sequence in a functional language and thus dispenses further explanation.



```
TestCase TC_Send_SMS_Message :
  Setup :
      description("Turn on the phone and go to the messaging application.");
      phone.turnOn();
      phone.goTo(MESSAGING_APPLICATION);
      phone.deleteAllMsg();
  Test :
      description("Create a SMS message.");
      SMS msg = phone.create(SMS_MESSAGE);
      msg.addContent("Test SMS message");
      msg.addDestination(MY_OWN_NUMBER)
      phone.setFields(msg);

      description("Send SMS message.");
      phone.send(msg);

      description("Verify the receiving of SMS message.");
      phone.openSMS(msg);
      phone.checkContent(msg);
  Cleanup :
      description("Delete the message and go back to the initial status.");
      phone.deleteAllMsg();
      phone.goTo(IDLE);
```

Fig. 4. An instance of a standard test script

```
TC_Send_SMS_Message =
 <setup, turnOn, goTo.MESSAGING_APP, delete.ALL_MESSAGES,
  test, create.SMS, send.SMS, verify.SMS,
  cleanup, delete.SMS, goTo.IDLE>
```

It is worth noting the similarity of the CSP event names with the elements in Fig. 4. For instance, the `Setup` method became the `setup` event. After the previous task is done in all test cases that form a given feature under test, we can build an overall model to capture all test cases in a single behavioural unit (CSP process). It will represent a specific feature of the phone that the test cases are supposed to cover. To give an idea of the previous effort, we present the generated CSP process from a set of formal test cases. Thus, initially we have three test cases: one for sending a SMS message

```
TC1 = <setup, turnOn, goTo.MESSAGING_APP,
        test, create.SMS, send.SMS, verify.SMS,
        cleanup, delete.SMS, goTo.IDLE>
```

another for a MMS message

```
TC2 = <setup, turnOn, goTo.MESSAGING_APP,
        test, create.MMS, send.MMS, verify.MMS,
        cleanup, delete.MMS, goTo.IDLE>
```

and the last one for an EMS message.

```
TC3 = <setup, turnOn, goTo.MESSAGING_APP,
        test, create.EMS, send.EMS, verify.EMS,
        cleanup, delete.EMS, goTo.IDLE>
```

The composition of the previous formal test cases (the test cases model) yields the following CSP process.

```
TEST_MODEL = setup -> turnOn -> goTo.MESSAGING_APP ->
              test ->
              (    create.SMS -> SENDING(SMS)
                [] create.EMS -> SENDING(EMS)
                [] create.MMS -> SENDING(MMS) )

SENDING(msg) = send.msg -> verify.msg ->
                cleanup -> delete.msg -> goTo.IDLE -> SKIP
```

The test cases model, or test model, is represented as a finite CSP process (see `TEST_MODEL`). This is always possible because a typical test case has not recursive calls, it only follows simple sequential execution flow (see the test cases `TC1`, `TC2`, and `TC3`) and always end in a `SKIP` process (or successful termination).

4.2 From Requirement to CSP Model

The requirements are described by use case descriptions, where each use case is divided in main and exception flows. Each flow has one or more steps described in natural language. Recall from Fig. 1, it shows a standard use case. Each use case must also be represented following the CSP notation. The task of modelling use cases as CSP processes can be found in [4]. It basically takes each step written in CNL and represents them as a CSP event (see [3, 4] for more details). The events are generated and organized following the flows contained in the use case. Finally, we have a CSP process that contains all generated events in their respective flows. In what follows we illustrate a use case written formally. The `UC_01` represents a formal use case generated from the use case of Fig. 2.

```
UC_01 = turnOn -> MAIN_FLOW
MAIN_FLOW = goTo.MESSAGING APP -> create.(SMS, NUMBER.5555555,
CONTENT.Test) ->
HANDLE((SMS, NUMBER.5555555, CONTENT.Test))

HANDLE(msg) = send.msg -> verify.msg ->
```

```
              ( EXCEPTION_FLOW_01(msg) [] EXCEPTION_FLOW_02(msg) )
EXCEPTION_FLOW_01(msg) =thereIsNot.msg ->checkOnMesg.NUMBER ->MAIN_FLOW
EXCEPTION_FLOW_02(msg) =thereIs.msg -> delete.msg -> goTo.IDLE -> UC_01
```

After each use case is translated to CSP, they must be composed together to form the use model, which is a unique behaviour unit (CSP process). The composition of use cases is out of the scope of this paper, so we assume that there is this model composed initially. A more detailed description about how to translate standard use cases to formal use cases and also how to compose formal use cases can be found in [4, 3]. The `REQUIREMENTS_MODEL` below illustrates a final (composed) requirements model generated from two use cases (similar to `UC_01`), where both are regarding to actions of sending messages (`SMS` and `EMS`).

```
REQUIREMENTS_MODEL = turnOn -> MAIN_FLOW
MAIN_FLOW = goTo.MESSAGING_APP ->
            ( create.SMS -> HANDLE(SMS) [] create.EMS -> HANDLE(EMS) )
HANDLE(msg) = send.msg -> verify.msg ->
              ( EXCEPTION_FLOW_01(msg) [] EXCEPTION_FLOW_02(msg) )
EXCEPTION_FLOW_01(msg) = thereIsNot.msg ->
                         checkOnMesg.PHONE NUMBER -> MAIN_FLOW
EXCEPTION_FLOW_02(msg) = thereIs.msg ->
                         delete.msg -> goTo.IDLE -> REQUIREMENTS_MODEL
```

In the previous requirements model (`REQUIREMENTS_MODEL`), note the generated choice between `create.SMS` and `create.EMS` in the external choice of the process `MAIN_FLOW`. Such choice well characterizes the unified use cases.

Similarly to the test model, the requirements model is also a CSP process, although it differs because the requirements model contains recursive calls. Due to recursive behaviour of CSP process the requirements model can generate infinite traces. The same does not apply to the test model as explained farther.

5 Defining the *merge* Operator

Our merge operator must behave similar to parallelism on common events and as external choice on distinct ones. Fig. 5 and Fig. 6 are illustrating our merging approach. Note that, the points `A`, `D` and `E` are common to both models (they are indeed our intersection points). Each one must be factorized as a single sequence in the final (resulting) model as represented in Fig. 6.

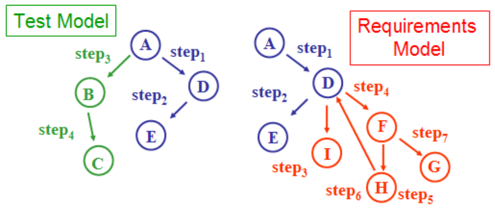


Fig. 5. Test and requirements models

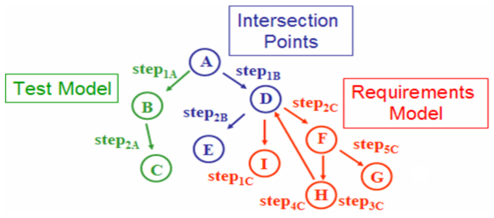


Fig. 6. Use of intersection points to unify models

According to Fig. 6 the unified model is composed by three distinct parts: the test model (nodes B and C), the intersection points (nodes A, D and E) and the requirements model (nodes F, G, H and I). The unified model of Fig. 6 is generated using the test model (left-side) and requirements model (right-side) of Fig. 5. It is important to note that, in real terms, in order to well define the intersection points the test and requirement models should describe the same set of functionalities (feature).

A requirements model, differently of the test case model, is based on recursive calls, that is, it has infinite behaviours. Thus, to compose both models into a single one we cannot use the functional features of CSP. Thus, we have two main ways of doing this: (i) by using a combination of CSP operators; (ii) by proposing a new CSP operator. In this paper we explore the latter alternative.

5.1 CSP *merge* Operator

As it was mentioned before, we define the merge operator using an approach based on operational semantics. This new CSP operator, ⊕ (the merge operator), is responsible for unifying CSP models. Fig. 7 shows the approach of updating the requirements model using the merge operator. Initially we have an out of date requirements model and an up to date test case model. The merge operator receives both models and performs the unification. The result of this unification effort is a new updated requirements model.

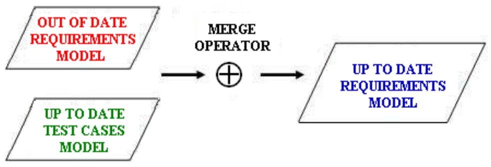

OUT OF DATE REQUIREMENTS MODEL
UP TO DATE TEST CASES MODEL
MERGE OPERATOR
⊕
UP TO DATE REQUIREMENTS MODEL

Fig. 7. Unifying CSP models: requirements and test cases

Our merge operator, as previously cited, must behave similar to parallelism on common events and as external choice on distinct ones. Thus, similar to Roscoe's [5] definition, we have rules to promote τ actions:

$$\frac{P \xrightarrow{\tau} P'}{P \oplus Q \xrightarrow{\tau} P' \oplus Q} \qquad \frac{Q \xrightarrow{\tau} Q'}{P \oplus Q \xrightarrow{\tau} P \oplus Q'} \qquad (1)$$

The merge operator also needs to handle distributed termination (see rules (2)), similarly to the rules of parallelism.

$$\frac{P \xrightarrow{\checkmark} P'}{P \oplus Q \xrightarrow{\tau} \Omega \oplus Q} \qquad \frac{Q \xrightarrow{\checkmark} Q'}{P \oplus Q \xrightarrow{\tau} P \oplus \Omega} \tag{2}$$

Once all arguments have terminated and become Ω, we can finish the whole process using the following rule:

$$\frac{}{\Omega \oplus \Omega \xrightarrow{\checkmark} \Omega} \tag{3}$$

There are two rules for ordinary visible events: one for common events (a parallel behaviour):

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \oplus Q \xrightarrow{a} P' \oplus Q'} \ (a \neq \tau) \tag{4}$$

and another for distinguishing events (an external choice behaviour):

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q'}{P \oplus Q \xrightarrow{a} P'} \ (a \neq b) \quad \frac{P \xrightarrow{a} P', Q \xrightarrow{b} Q'}{P \oplus Q \xrightarrow{b} Q'} \ (a \neq b) \tag{5}$$

6 Unifying CSP Models

The strategy of unifying CSP models has the support of some tools: the *Use Model Generator* [4] (Requirements to CSP Models), the *TCRev* [9] (Test Cases to CSP Models and Unifying Models) and the *CSP2CNL* [10] (CSP Models to Requirements). These tools have been developed to be part of a major tool, called *TaRGeT*. They are not yet working attached to the TaRGeT, but the idea is that they will be accessed from the TaRGeT's menu, as illustrated in Fig. 8. Thus, all CSP formalism (models) will be hidden from the user. There is no need to have a CSP background, once the user will only deal with the Requirements and Test Cases documents written in English (controlled natural language).

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

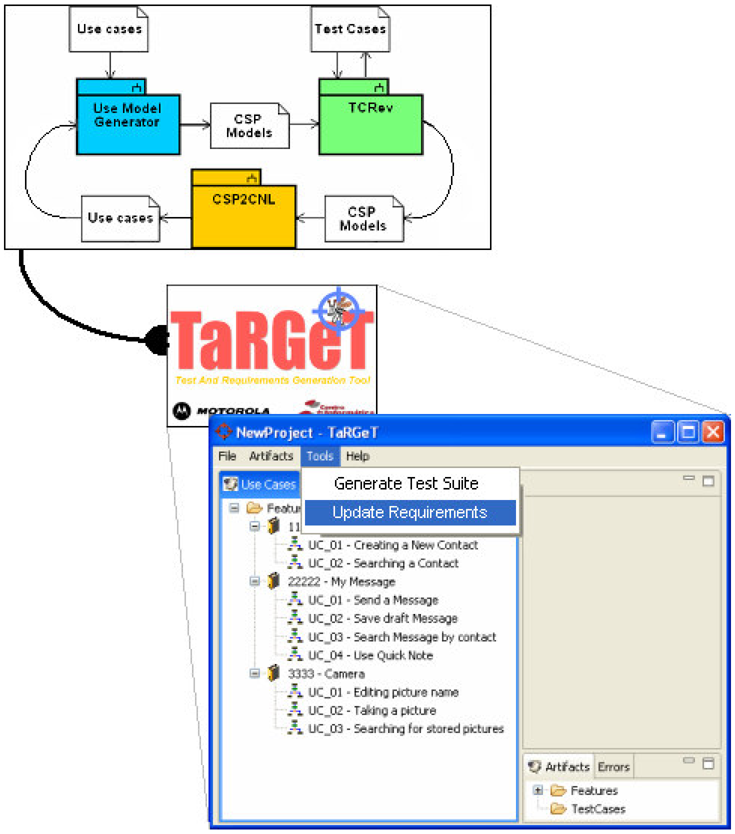


Fig. 8. Tools integration with TaRGeT

As the tools are not yet working attached to the TaRGeT, in this paper we use each tool separately. First, the *Use Model Generator* translates a requirements document into its CSP model and the *TCRev* translates the test cases documents related to those requirements into their CSP models. Then, the *TCRev* with both models, test and requirements models specified in CSP, can start the unifying step. It consists in updating the original requirements model using the test model. Thus, the result of this unification effort is a single CSP model, which contains information about the original requirements (out of date) as well as that provided by test cases (up to date). Then, with the resulting CSP model, it is now possible to translate it to an updated requirements document written in English. For this effort, we use the *CSP2CNL* tool.

In order to present our unifying strategy in the context of an industrial project, we are considering test cases and requirements that deal with a specific functionality of mobile phone. Thus, test cases as well as requirements which describe similar functionalities are grouped in specific features. For instance, a set of test cases related to sending and receiving messages are grouped in a same feature, whereas test cases related to multimedia (such as, playing audio and video files, and others) are grouped in another feature. Our case study deals only with messaging based features. In our context, the messaging feature deals with three possible types of messages: SMS, EMS, and MMS messages.

The first step is to model the set of test cases as a unique formal model. Thus, for simplification we consider the test model of Section 4.1. After modelling the test cases as a single behavioural model (`TEST_MODEL`), we can use the final model to update the requirements formal model. This is simply achieved by applying our proposed merge operator in the test and requirements models. As an example of an out of date requirements model we have the same requirements models shown in Section 4.2. The `REQUIREMENTS_MODEL` is out of date because it deals only with two kinds of messages (SMS and EMS), it does not characterize MMS messages. Thus, we define the process `UPDATED_USE_MODEL` in terms of the previous formal models and our proposed *merge* operator ⊕, as follows:

```
UPDATED_USE_MODEL = REQUIREMENTS_MODEL ⊕ (TEST_MODEL \ TEST_ALPHA)
```

where `TEST_ALPHA = {|setup, test, cleanup|}` is the set of specific test events, which are not found in the requirements model and are exclusively related to testing purposes.

By expanding the process `UPDATED_USE_MODEL` using the operational semantics of the merge operator, we can obtain the following explicit CSP process version.

```
UPDATED_USE_MODEL = turnOn -> MAIN_FLOW
MAIN_FLOW = goTo.MESSAGING_APP ->
            ( create.SMS -> HANDLE(SMS) [] create.EMS -> HANDLE(EMS) []
              create.MMS -> HANDLE(MMS))
HANDLE(msg) = send.msg -> verify.msg ->
              ( EXCEPTION_FLOW_01(msg) [] EXCEPTION_FLOW_02(msg))
EXCEPTION_FLOW_01(msg) = thereIsNot.msg ->
                         checkOnMesg.PHONE_NUMBER -> MAIN_FLOW
EXCEPTION_FLOW_02(msg) = thereIs.msg -> delete.msg ->
                         goTo.IDLE -> UPDATED_USE_MODEL
```

In the previous specification we have the final updated model, which it was added a new branch to handle MMS messages (`[] create.MMS -> HANDLE(MMS)`). This information, not found in the original requirements model, is provided from the test model. Furthermore, the work reported in [11] translates back the previous CSP representation into a CNL use case description.

7 Conclusion

In this paper, we present an approach of unifying models using the process algebra CSP. Our models are originated from test cases and requirements, and the main goal is to update the original requirements formal model with the updated information provided from the test cases model.

We propose a new CSP operator, called *merge*, by using the CSP operational semantics. The merge operator, defined via firing rules, is responsible for updating the original specification of the system, which is, in general, incomplete and out of date. The effort in this direction was to propose a composition mechanism to deal with infinite traces. Finite traces are common to test cases because they do not have recursive calls. Requirements, on the other hand, are more general, and thus it is common to find recursive calls, which allows the generation of infinite traces.

As future work we intend to verify and prove some properties of the merge operator, such as, associativity, commutativity, and others. Another future work is to implement the merge operator using parallel composition and functional aspects of CSP [12]. This is an attempt to define the merge behaviour in syntactical terms instead of semantical ones, which was presented here. Both merge descriptions (syntactical and semantical) are possible solutions, but it is necessary to consider the benefit of each one. In syntactic terms we have more power for expressing the merge behaviour, while using a semantical approach we gain more abstraction on the unification.

We are also working toward the integration of the cited tools. The aim is to integrate the *Use Model Generator* [4], the *TCRev* [9] and the *CSP2CNL* [10] tools in the TaRGeT tool, so that the transformation of test cases and the unification with the use cases to generate an up to date final requirements documents could be executed automatically, without interruptions and users' interaction. An additional effort is to link the TaRGeT tool with FDR [7]. In order to establish this link we intend to reuse FDR features as much as possible. Thus, it will be possible to check system properties of the generated formal models (test and requirements models, and also the unified model) in a complementary and interactive way.

8 References

1. I. Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2000.
2. C. J. Fillmore. *Frame semantics and the nature of language*. In Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech, Vol. 208: 20-32, 1976.
3. D. Leitão, D. Torres, and F. Barros. *NLForSpec: Translating natural language descriptions into formal test case specifications*. Technical report (TR-0606), Jun 2006.
4. G. Cabral and A. Sampaio. *Formal specification generation from requirement documents*. In Proceedings of III Brazilian Symposium on Formal Methods, pages 217–232, Natal, RN, Brazil, 2006.
5. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall (Pearson), 1997.
6. B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, University of Oxford, 1998.
7. Formal Methods (Europe) Ltd. *FDR User's Manual*, version 2.28 edition, 1997.
8. Formal Methods (Europe) Ltd. *PROBE Users Manual*, version 1.25 edition, 1998.
9. C. F. Sousa. *Modelling and Integrating Formal models: from Test Cases and Requirements Models*. Master's thesis, Universidade Federal de Pernambuco, Recife, PE, Brasil, 2006.
10. G. Peres. *Automatic English Requirements Generation from CSP Models*. Graduation work, Universidade Federal de Pernambuco, Recife, PE, Brasil, 2007.
11. G. Peres and A. Mota. *A Tool to Translate CSP Models into English Requirements*. II Encontro Brasileiro de Teste de Software, 2007 (http://ebts2007.cesar.org.br).
12. W. Mesquisa, A. Sampaio, and A. C. V. de Melo. *A strategy for the formal composition of frameworks*. SRFM, pages 404–413, 2005.