Guided Test Generation from CSP Models

Sidney Nogueira[1,2], Augusto Sampaio[1], and Alexandre Mota[1]

[1] Centro de Informática, Universidade Federal de Pernambuco
Caixa Postal 7851 - 50732-970 - Recife/PE - Brazil
[2] Mobile Devices R&D Motorola Industrial Ltda,
Rod SP 340 - Km 128, 7 A - 13820 000 - Jaguariuna/SP - Brazil

Abstract. We introduce an approach for the construction of feature test models expressed in the CSP process algebra, from use cases described in a controlled natural language. From these models, our strategy automatically generates test cases for both individual features and feature interactions, in the context of an industrial cooperation with Motorola Inc., where each feature represents a mobile device functionality. The test case generation can be guided by test purposes, which allow selection based on particular traces of interest. More generally, we characterise a testing theory in terms of CSP: test models, test purposes, test cases, test execution, test verdicts and soundness are entirely defined in terms of CSP processes and refinement notions. We have also developed a tool, ATG, which mechanises the entire generation process.

1 Introduction

Some of the main problems of effective testing is the selection of a good set of test cases and its automation [7], aiming at making the process more agile, less susceptible to errors and less dependent on human interaction. Formal notations like Finite State Machines (FSM) and Labelled Transition Systems (LTS) can provide accurate models for software that can be processed by tools that automatise the test design activity. There are several test generation approaches that use such models, as, for instance, [5, 21].

While LTS and FSM are the main models used as basis to automate test generation, they are very concrete models and often adopted as the operational semantics of more abstract process algebras like CSP [15], CCS [12] and LOTOS [11]. Contrasting with operational models, process algebra models can naturally evolve to incorporate additional requirements; the operators of a process algebra also allows complex models to be built from simpler ones, compositionally. Test generation can take advantage of this modular structure, and the approach can be formalised in terms of the process algebra semantic models.

Particularly, CSP is the standard formalism of the Brazil Test Center (BTC) research project [16], a cooperation between the Federal University of Pernambuco and Motorola Inc., in the context of testing embedded software that run over mobile phones. The rich repertoire of CSP operators are used to model individual features (mobile device functionalities) as well as several patterns of feature interaction. The CSP models are automatically constructed [2] from use cases described in a domain specific language [19, 9] (a small subset of English with a fixed grammar) for mobile applications.

The main contribution of this paper is a uniform strategy for generating test cases from CSP models. Instead of devising explicit generation algorithms (for instance, to deal separately with individual features and with feature interaction), our approach is based on using the CSP model checker (FDR) [10] in background. Test scenarios are incrementally generated as counter-examples of refinement verifications using FDR. Test selection is captured by CSP processes that describe the properties of interest, based on the concept of test purpose [8]; writing test purposes can also benefit from the expressiveness of CSP. The refinement relations submitted to FDR involves the original model and an annotated model obtained from the parallel composition of the original model and the test purposes.

In our testing theory, we consider as test hypothesis that the class of implementations to be tested can be specified by some CSP process [3]. We introduce an implementation relation, **cspio**, which defines the set of observations considered in testing: the implementation must produce a subset of the outputs for the inputs that are specified; although CSP does not differentiate input and output events, we make this distinction using separate input and output alphabets. Moreover, assuming that implementations are input enabled (accept all inputs in the alphabet) and eventually produce an output for a given input, we prove that test cases are sound in the sense that they do not reject correct implementations according to **cspio**. All the elements of our approach are entirely characterised in terms of CSP processes and refinement notions.

Some previous approaches have addressed test generation [13, 17, 3] in the context of CSP. The focus of [17] is the formalisation of conformance relations, while [13, 3] also consider the generation of infinite test sets. Nevertheless, these works do not distinguish input and output events nor address test purposes as a selection mechanism. Moreover, they do not provide tool support for automatic test case generation.

Section 2 presents our application domain (mobile device software). Section 3 shows how CSP is used to construct test models, both for individual features and for feature interaction. Section 4 addresses test scenario generation based on process refinement, and test selection based on test purposes is the subject of Section 5. Section 6 introduces our CSP characterization of conformance testing and show how to obtain sound test cases from a set of test scenarios. Section 7 presents details about the ATG tool and the results of a case study performed with such tool. The final section briefly discusses tool support and considers related and future work.

2 Application Domain

The development process of mobile phone software follows an iterative approach, where sets of functionalities (known as features) are incrementally considered in each development cycle. An example of a feature is the set of requirements for sending a multimedia message. In general, new features are developed and tested, firstly, in isolation, and later integrated with other features, giving rise to feature interactions.

The main inputs for the automatic test generation approach in the BTC project are use case documents that describe the behaviour of the features to be tested, and the output is a test case suite suitable for manual execution. Input and output templates obey a Controlled Natural Language (CNL) standard [19, 9] that can be translated to and from CSP.

In what follows, we overview the use case documents, see Figures 1 and 2.

From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Go to Message Center.		"Important Messages" folder is displayed.
2M	Go to "Inbox".		All Inbox Messages are displayed.
3M	Scroll to a message.		Message is highlighted.
4M	Go to Context Sensitive Menu.		"Move to Important Messages" option is displayed.
5M	Select "Move to Important Messages" option.	Message storage is not full.	"Message moved to Important Messages folder" is displayed.

Fig. 1. Main flow

From Step: 5M
To Step: END

Step Id	User Action	System State	System Response
1A	Select "Move to Important Messages" option.	Message storage is full.	"Message storage is full" is displayed. "Clean Up request" is displayed.
2A	Perform clean up.		"Message moved to Important Messages folder" is displayed.

Interaction Point: 2M

Step Id	User Action	System State	System Response
1I	Select store status option information.		Storage status dialog is displayed.
2I	Dismiss the storage status dialog.		Storage status dialog is closed.

Fig. 2. Alternative and interaction flows

Feature Use Cases A feature use case has a set of interconnected flows (main, alternative and exception); each flow is a sequence of steps, and each step has an identifier (Id) that is used for referencing (use cases can be shared by different features and documents). Features and use cases also have unique identifiers. The complete reference for a step has the form FEATURE_ID#UC_ID#STEP_ID. Moreover, each flow step specifies an user input action (User Action column), the expected system output in the System Response column, and the (optional) condition required to produce the expected system output (System State column). Figure 1 shows the main flow of the use case of the Important Messages Feature. Such a flow specifies the sequence of actions that the user must perform to move a message from the Inbox to the Important Messages Folder. For instance, in step 5M, to get a message dialog that confirms the success of moving a message, the memory storage must not be full.

The fields 'From Step' and 'To Step' are used to indicate the set of steps from where the flow must start and to where it must continue. As a default, the main flow uses the constants START (no previous step) and END (no subsequent step) for these fields. Alternative flows are simply defined by characterising where (From Step) they can assume control and where they must resume (To Step), with respect to the flows they are referencing. Figure 2 (top) shows a possible alternative flow for the Important Messages use case. It specifies that, after step 4M of the Feature main flow, if the memory storage is full, the selected message is not moved because a clean up action is requested. After the clean up the message is moved to the Important Messages Folder and the alternative flow finalizes. The exception flows are similar to alternative flows, except for representing exceptional behaviours.

Feature Interaction Feature interactions are extensions of feature use cases by simply introducing interaction points. Using the field Interaction Point one can indicate a set of steps from which the interactive flow can assume control and resume control to the next step in the original flow. Figure 2 (bottom) shows the specification of an interactive flow that can interact after step 1M of the main flow. This interaction specifies that

after the main action "Go to Message Center", the feature can continue its main flow or verify the message storage status (interaction flow), and then continue the main flow from step 2M.

3 Test Models as CSP Processes

A process is the central element of a CSP specification. Processes can offer events from Σ (the set of possible events) to establish communication with the environment or with other processes. The alphabet of a CSP process, say P, is the set of events it can communicate, say α_P, where $\alpha_P \subseteq \Sigma$. Furthermore, the primitive process $Stop$ specifies a broken process (deadlock), and the primitive $Skip$ a process that communicates an event $\checkmark$ and terminates successfully.

Although there is no semantic distinction between input and output events in CSP, we consider that Σ is split into three disjoint sets of events: inputs Σ_i, outputs Σ_o and conditionals Σ_c. In our application domain, input events represent user actions, the output events model system responses, and conditional events abstract the system internal state. Then, $\Sigma = \Sigma_i \cup \Sigma_o \cup \Sigma_c$. Similarly, the alphabets of the processes follow the same structure: $\alpha_P = \alpha_{Pi} \cup \alpha_{Po} \cup \alpha_{Pc}$.

The rest of this section shows how CSP operators are used to build test models for our application domain. The operators are introduced by demand.

Modelling Individual Features Basic CSP operators as prefix and external choice are suitable to model feature use cases. The CSP prefix operator $P = ev \rightarrow Q$ specifies that event ev is communicated by P, which then behaves as the process Q. The external choice operator $P = Q \,\Box\, R$ indicates that the process P can behave as Q or R; the choice is made by the environment.

As an example, applying the translation approach presented in [2] to the main and alternative flows of the use case of Important Messages Feature (Figures 1 and 2) we obtain the model specified as follows.

$$
\begin{aligned}
UC_1 = {} & goToMsgCenter \rightarrow IMFolderIsDisp \rightarrow goToInbox \rightarrow inboxMsgsDisp \rightarrow \\
& scrollToAMsg \rightarrow msgHighlighted \rightarrow goToCSM \rightarrow moveToIMOptDisp \rightarrow \\
& selMoveToIMOpt \rightarrow (UC_{11} \,\Box\, UC_{12}) \\
UC_{11} = {} & msgStoIsNotFull \rightarrow msgMovedToIMDisp \rightarrow Skip \\
UC_{12} = {} & msgStoIsFull \rightarrow cleanUpReqDisp \rightarrow \\
& performCleanUp \rightarrow msgMovedToIMDisp \rightarrow Skip
\end{aligned}
$$

The process UC_1 specifies the main use case flow (Figure 1) up to Step $4M$ (event $selMoveToIMOpt$). From this point, it behaves as the choice $UC_{11} \,\Box\, UC_{12}$. The process UC_{11} specifies Step $5M$ of the main flow, and UC_{12} the behaviour of the alternative flow (Figure 2). Both main and alternative flows finish with success (behave as $Skip$).

Modelling Feature Interactions Now we show an approach to capture feature interactions using CSP, by combining the CSP processes that specify interaction flows with the processes that specify main, alternative and exception flows.

Consider the CSP notation $P \setminus s$ that defines a process which behaves like P communicates all its events, except the events that belong to s, which become internal (invisible): $\setminus$ stands for the hiding operator. The process $P \,|[\, X \,]|\, Q$ stands for the

generalised parallel composition of the processes P and Q with synchronisation set X. This expression states that P and Q must synchronize on events that belong to X. Each process can evolve independently for events that are not in X.

Figures 3 and 4 give a graphical overview on how to model feature interaction by using CSP parallel composition. The top of Figure 3 represents the use case case process UC_1 modified with the insertion of one interaction point (the events $beginI.1$ and $endI.1$ are control events). The bottom of the same figure illustrates the interaction process for the interaction flow of Figure 2. Finally, putting these processes in parallel with synchronization set $\{beginI.1, endI.1\}$ and hiding this set, we obtain the model exhibited in Figure 4. The transition labelled as $\_tau$ in Figure 4 denotes invisible event and appears in the resulting model to allow the interaction to occur optionally; this is further discussed in the sequel.

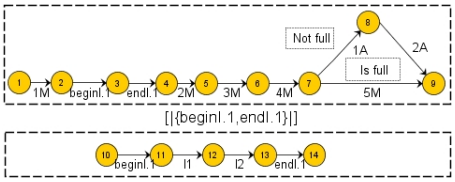

Fig. 3. Marked use case and interaction

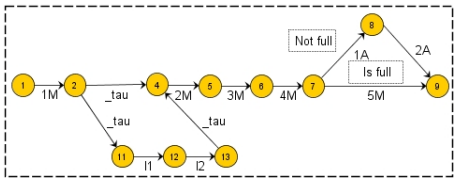

Fig. 4. Feature interaction model

The rest of this section materializes the graphical modelling of interaction shown in Figures 3 and 4 motivated above in terms of CSP. Consider the indexed external choice construction of CSP $\Box\, x : A \bullet F(x)$, where x can be a value or an event from set A, and $F(x)$ any CSP term involving x. This construction behaves as the external choice $F(x_1) \,\Box\, F(x_2) \,\Box\, ... \,\Box\, F(x_n)$ for $A = \{x_1, x_2, ..., x_n\}$.

We define the auxiliar process $InteractionPoint$ that is used to introduce control points in the use case processes that are affected by interactions.

$$InteractionPoint(indices) = Skip \,\Box\, (\Box\, i : indices \bullet beginI.i \rightarrow endI.i \rightarrow Skip)$$

The parameter $indices$ is a set of interaction identifiers used to characterise which interactions are allowed in the same point of a given use case. The events from $IntControl = \{beginI.i, endI.i\}$ are used to specify such points. For each $i \in indices$ the process $InteractionPoint$ offers a choice with the prefix $begin.i \rightarrow end.i \rightarrow Skip$. Furthermore, the external choice with $Skip$ allows the original flow to perform without any interaction.

The process UC_1' in Figure 3 (top) is the process UC_1 from the previous section modified by the insertion of an interaction point after the step 1M. In UC_1' the processes UC_{11} and UC_{12} remain unchanged because there are no interactions for them.

$$\begin{aligned} UC_1' = {} & goToMsgCenter \rightarrow IMFolderIsDisp \rightarrow Skip;\ InteractionPoint(\{1\}); \\ & goToInbox \rightarrow inboxMsgsDisp \rightarrow scrollToAMsg \rightarrow msgHighlighted \rightarrow \\ & goToCSM \rightarrow moveToIMOptDisp \rightarrow selMoveToIMOpt \rightarrow (UC_{11} \,\Box\, UC_{12}) \end{aligned}$$

The parameter {1} for the process $InteractionPoint$ above is the index for the interaction Figure 2 whose CSP specification is the process

$$STORAGE\_STATUS = selStoStaOpt \rightarrow stoStaDiaDisp \rightarrow dismStoStaDia \rightarrow stoStaDiaClosed \rightarrow Skip$$

In addition, we define the auxiliar process I that is similar to $InteractionPoint$ except that it handles a unique control point (instead of a set), and between the $beginI.index$ and $endI.index$ events it includes the interaction flow itself.

$$I(index, interaction) = Skip \,\Box\, (beginI.index \rightarrow interaction;\ endI.index \rightarrow Skip)$$

Finally, the CSP interaction specification of the use case flows of Figures 1 and 2 is

$$UC_1\_I = (UC'_1 \,[\![\, IntControl \,]\!]\, I(1, STORAGE\_STATUS)) \setminus IntControl$$

where the process $I(1, STORAGE\_STATUS)$ (represented in Figure 4, at bottom) allows the flow $STORAGE\_STATUS$ to occur in the point where $InteractionPoint(\{1\})$ is included in UC'_1. The events of $IntControl$ are hidden from the model since they only play the role of control events. Figure 4 shows a graphical view of the process UC_1_I. Note that the original flow can be interrupted at point 2, where the subflow demarcated by 11, 12 and 13 is the interruption flow.

The notation $\mu\, X.F(X)$ stands for a nameless recursive CSP process. For the general case, the CSP process below specifies the feature interaction model for a set of independent feature model processes $\{UC'_1, ..., UC'_N\}$ with a set of independent interactions $\{I(1, int_i), ..., I(n, int_n)\}$ that can occur in any point of the feature models.

$$UC\_I = (\,\Box\, k : \{1..N\} \bullet UC'_k \,[\![\, IntControl \,]\!]\, \mu\, X.\, \Box\, i : \{1..n\} \bullet I(i, int_i);\ X) \setminus IntControl$$

On the left-hand side of the parallelism, the use cases are modelled as the external choice of the respective processes; each process UC'_k ($1 \leq k \leq N$) stands for the use case UC_k modified with the insertion of interaction points. On the right-hand side, there is a choice among the possible interactions $I(i, int_i)$ ($1 \leq i \leq n$) that recurs after successful termination. This recursion allows the run of any interaction whenever the respective interaction points are reached in the use cases.

A more elaborate model of feature interaction can be achieved using the interleaving operator of CSP. We can define that the occurrences of the process $I(i, int_i)$ are interleaved, and that each occurrence is itself recursive. Similarly, the use case models can as well be combined using interleaving. This allows multiple interactions to occur simultaneously, and is useful in the context of concurrent features.

Semantic Models for CSP Trace semantics is the simplest model for a CSP process. The traces of a process P, given by $traces(P)$, correspond to the set of all possible sequences (even infinite) of events P can communicate. For the process $Stop$, $traces(Stop) = \{\langle\rangle\}$, and for $Skip$, $traces(Skip) = \{\langle\rangle, \langle\checkmark\rangle\}$. For prefix, $traces(a \rightarrow P) = \{\langle\rangle, \langle a\rangle\} \cup traces(P)$. Let $P1$ and $P2$ be two CSP processes, then $traces(P1 \,\Box\, P2) = traces(P1) \cup traces(P2)$. A complete definition for all CSP operators can be found in [15].

It is possible to compare the traces semantics of two processes by refinement verification using the FDR [10] tool. A process Q refines the process P in the traces model, say $P \sqsubseteq_\tau Q$, if and only if $traces(P) \supseteq traces(Q)$. Otherwise, FDR yields a trace (the shortest counter example), say ce, such that $ce \in traces(Q)$ but $ce \notin traces(P)$. For instance, $UC_{11} \Box UC_{12} \sqsubseteq_\tau UC_{12}$ holds, since $traces(UC_{11} \Box UC_{12}) \supseteq traces(UC_{12})$. However, the relation $Skip \sqsubseteq_\tau Skip;\ accept.1 \rightarrow Stop$ does not, since $\langle accept.1 \rangle \in traces(accept.1 \rightarrow Stop)$ but $\langle accept.1 \rangle \notin traces(Skip)$. Thus, the trace $\langle accept.1 \rangle$ is a counter example.

Structuring the process UC_I as explained previously, we have that $UC\_I \sqsubseteq_\tau UC$ holds: the traces of the use case model without interactions is included in the traces of the interaction model. For instance, $UC_1\_I \sqsubseteq_\tau UC_1$ holds.

Other more elaborate semantic models of CSP are the failures and the failures-divergences models. The former captures deadlock situations, whereas the latter captures livelocks as well. See [15] for further details.

4 Test Scenario Generation

Given a test model S and a safety property Φ, we can obtain the traces of S that satisfy Φ. We call these traces test scenarios, say ts, when Φ describes some test selection criteria. A test scenario is the central element used to construct a CSP test case. This Section shows how to generate test scenarios as the counter examples of refinement verifications.

Consider the set $MARK = \{accept.n\}$ for $n \in \mathbb{N}$, the alphabet of mark events used in our test generation approach. Let S be the process that specifies the model we want to select tests from, then we define S' to be S with the addition of mark events after test scenarios that satisfies Φ. The idea is to perform refinement verifications of the form $S \sqsubseteq_\tau S'$ that generate the test scenarios as counter examples. Consider that $s_1 \frown s_2$ indicates the concatenation of sequences s_1 and s_2, and $\langle ev \rangle$ a sequence containing the element e. Then, S' is defined in such a way that for all test scenarios $ts \in traces(S)$ that satisfies Φ, there is a trace $ts \frown \langle m \rangle \in traces(S')$, such that $m \in MARK$ and $MARK \cap \alpha_S = \emptyset$. As a consequence $ts \frown \langle m \rangle \notin traces(S)$, so the relation $S \sqsubseteq_\tau S'$ does not hold and the counter examples are traces of the form $ts \frown \langle m \rangle$. The shortest test scenario (counter example), say ts_1, is retrieved by FDR when $S \sqsubseteq_\tau S'$ does not hold.

To illustrate the proposed test scenario generation approach, we show how to generate a set of test scenarios ($ts \in traces(S)$) that lead the test model to successful termination. Consider the CSP process $P;\ Q$ that behaves like P until it terminates successfully, when the control passes to Q. Consider the process $ACCEPT(id) = accept.id \rightarrow Stop$ that is used to mark test scenarios by communicating the mark event $accept.id$ ($accept.id \in MARK$). Thus, we define S' as the process $(S;\ ACCEPT(i))$. This process inserts marks ($accept.i$) after each successful termination of S. As a consequence, the verification of relation ($S \sqsubseteq_\tau S'$) yields as counter examples the test scenarios that lead the specification to successful termination (if they exist).

For example, checking the relation $UC_1 \sqsubseteq_\tau UC_1;\ ACCEPT(1)$ using FDR results in the shortest counter example, as displayed below.

$UC_1\_ts_1 = \langle goToMsgCenter, IMFolderIsDisp, goToInbox, inboxMsgsDisp, scrollToAMsg, msgHighlighted, goToCSM, moveToIMOptDisp, selMoveToIMOpt, msgStoIsNotFull, msgMovedToIMDisp, accept.1\rangle$

The above trace (ignoring the marking event $accept.1$) is the shortest successful termination test scenario to UC_1. It corresponds to the main use case flow of the Important Messages Feature (Figure 1).

To obtain from S subsequent test scenarios lengthier than a test scenario ts_1, we use the function $Proc$ that receives as input a sequence of events and generates a process whose maximum trace corresponds to the input sequence. For instance, $Proc(\langle a, b, c\rangle)$ yields the process $a \rightarrow b \rightarrow c \rightarrow Stop$. The reason for using $Stop$, rather than $Skip$, is that $Stop$ does not generate any visible event in the traces model, while $Skip$ generates the event $\checkmark$.

The second counter example is generated from S using the previous refinement, but the process formed by the counter example ts_1 ($Proc(ts_1)$) as an alternative to S on the left-hand side. The second test scenario can then be generated as the counter example to the refinement $S \ \Box\ Proc(ts_1) \sqsubseteq_\tau S'$. As $traces(S \ \Box\ Proc(ts_1))$ is equivalent to $traces(S) \cup ts_1$, ts_1 cannot be a counter example of the second refinement iteration. Thus, if the refinement does not hold again, then we have ts_2 as the counter example.

The iterations can be repeated until the desired set of test scenarios is obtained (for instance, a fixed number of tests is generated). In general, the $n + 1^{th}$ test scenario can be generated as a counter example of the following refinement.

$$S \ \Box\ Proc(ts_1) \ \Box\ Proc(ts_2) \ \Box\ ... \ \Box\ Proc(ts_n) \sqsubseteq_\tau S' \quad (1)$$

Continuing the selection of successful termination traces of UC_1, checking the relation $UC_1 \ \Box\ Proc(UC_1\_ts_1) \sqsubseteq_\tau UC_1;\ ACCEPT(1)$ yields a second counter example.

$UC_1\_ts_2 = \langle goToMsgCenter, IMFolderIsDisp, goToInbox, inboxMsgsDisp, scrollToAMsg, msgHighlighted, goToCSM, moveToIMOptDisp, selMoveToIMOpt, msgStoIsFull, cleanUpReqDisp, performCleanUp, msgMovedToIMDisp, accept.1\rangle$

The above trace is another successful termination test scenario for UC_1. It corresponds to the alternative flow of the Important Messages (Figure 2). Finally, since there is no more successful termination scenarios to generate from UC_1, the following refinement $UC_1 \ \Box\ Proc(UC_1\_ts_1) \ \Box\ Proc(UC_1\_ts_2) \sqsubseteq_\tau UC_1;\ ACCEPT(1)$ holds.

This strategy applies both to feature models and to feature interaction models, introduced in the previous section.

5 Test Scenario Selection

Although successful termination can itself be used as a selection criteria, as illustrated in the previous section, shows a more general strategy for selecting a set of test scenarios from a test model S based on the concept of a test purpose TP, described as a CSP process.

A CSP test purpose is based on the notion introduced in [8]: a test purpose is a partial specification describing the characteristics of the desired tests. The definition below formalizes the concept.

Definition 1 *Let TP and S be CSP processes. The process TP is a test purpose for S if it is deterministic behaviour and* $\forall\, ts \frown \langle m \rangle : traces(TP) \bullet ts \in traces(S) \wedge m \in MARK$.

A TP must be deterministic to avoid the selection of inconsistent test scenarios. The other relevant property of a TP is that its traces (excluding the mark event) must be traces of the specification model. To ease the task of writing TP in CSP following Definition 1, we provide a set of primitive processes that can be combined to design possibly complex test purposes. Table 1 presents the names and parameters of the primitives (left-hand column) and their definitions (right-hand column).

Table 1. Constructors for test purposes

Process Name	Process Specification
$ACCEPT(id)$	$accept.id \rightarrow Stop$
$REFUSE(id)$	$refuse.id \rightarrow Stop$
$ANY(evset, next)$	$\Box\ ev : evset \bullet ev \rightarrow next$
$NOT(\alpha_S, evset, next)$	$ANY(\alpha_S - evset, next)$
$MATCH(\alpha_S, evset, next, init)$	$ANY(evset, next) \,\Box\, NOT(\alpha_S, evset, init)$
$MATCHS(\alpha_S, s, next, init)$	$if (s = \langle\rangle)\ then\ next$ $else$ ($ANY(\{\text{head}(s)\},$ $MATCHS(\alpha_S, \text{tail}(s), next, init)$) $\Box$ $NOT(\alpha_S, \{\text{head}(s)\}, init)$)
$EXCEPT(\alpha_S, evset, next, init)$	$MATCH(\alpha_S, evset, init, next)$
$UNTIL(\alpha_S, evset, next)$	$RUN(\alpha_S - evset)$ $\triangle\ ANY(evset, next)$

The process $ACCEPT(id)$ was already explained in the previous section. We introduce a similar process $REFUSE(id) = refuse.id \rightarrow Stop$ that marks refusal test scenarios by communicating the special event $refuse.id$. It is used to stop the test selection ignoring useless test scenarios and saving computational resources.

The primitive $ANY(evset, next) = \Box\ ev : evset \bullet ev \rightarrow next$ performs basic selection. It selects the events offered by the specification that belong to $evset$. If any of these events can occur, it behaves as $next$. Otherwise, it deadlocks. The process $NOT(\alpha_S, evset, next)$ is complementary to ANY. It selects from the events offered by the specification those that belong to $\alpha_S - evset$.

The primitive $MATCH(\alpha_S, evset, next, init)$ is used to select from the events offered by the specification those that belong to $evset$. If it is the case, it behaves as $next$.

Otherwise, it behaves as $init$. The parameter $init$ indicates the initial state of a test purpose. Whenever the specification offers events that do not match the test purpose objective, the test purpose process restarts the selection, behaving as $init$. Consider the functions $head(s)$ and $tail(s)$ that yields the head and the tail of the sequence s. The primitive process $MATCHS(\alpha_S, s, next, initial)$ is an extension of $MATCH$. It is used to select a sequence s from the specification. If this is possible, it behaves as $next$. Otherwise, it behaves as $init$.

The process $EXCEPT(\alpha_S, evset, next, initial)$ is complementar to $MATCH$. If the offered events belong to $\alpha_S - evset$, it continues the selection, otherwise, it restarts and behaves as $init$.

Consider the process $RUN(s) = \Box\ ev : s \bullet ev \rightarrow RUN(s)$ that continuously offers the events from the set s, and $P\ \triangle\ Q$ which indicates that Q can interrupt the behaviour of P if an event offered by Q is communicated. The process $UNTIL(\alpha_S, evset, next) = RUN(\alpha_S - evset)\ \triangle\ ANY(evset, next)$ selects all sequences offered by the specification events until it engages on some event that belongs to $evset$.

The following is an example of a test purpose TP_1 that is used to select scenarios from UC_1. The objective of TP_1 is to select from UC_1 test scenarios whose final output is a message confirming that the selected important message is moved to the folder ($msgMovedToIMDisp$), and at some point before the user has performed a cleanup action ($performCleanUp$).

$$TP_1 = UNTIL(\alpha_{UC_1}, \{performCleanUp\}, \\ UNTIL(\alpha_{UC_1}, \{msgMovedToIMDisp\}, ACCEPT(1)))$$

The process TP_1 offers the events of α_{UC_1} until it engages on $performCleanUp$. Next, it offers the events of α_{UC_1} until it engages on $msgMovedToIMDisp$, when it behaves as $ACCEPT(1)$ that inserts the mark event $accept.1$.

Based on the test scenario generation approach from the previous section, one can select test scenarios for a given CSP test purpose TP by defining the process S' (here referred to as $PP(S, TP)$) as the *parallel product* of S with a test purpose TP with synchronisation set α_S: $PP(S, TP) = S\ |[\ \alpha_S\ ]|\ TP$. The process TP synchronises in all events offered by S until the test purpose that follows Definition 1 matches a test scenario, when TP communicates an event $mark \in MARKS$. At this point, the process TP deadlocks, and consequently $PP(S, TP)$ deadlocks as well. This makes the parallel product to produce traces $ts \frown \langle mark \rangle$, where ts are the test scenarios. If S does not contain scenarios specified by TP, no mark event is communicated, the parallel product does not deadlock and the relation $S \sqsubseteq_\tau PP(S, TP)$ holds.

Considering again our example, the shortest test scenario from UC_1 that matches the test purpose TP_1 is obtained from a counter example of the relation $UC_1 \sqsubseteq_\tau PP(UC_1, TP_1)$, where $PP(UC_1, TP_1) = UC_1\ |[\ \alpha_{UC_1}\ ]|\ TP_1$. The counter example is given below.

$$UC_1\_TP1\_ts_1 = \langle goToMsgCenter, IMFolderIsDisp, goToInbox, inboxMsgsDisp, \\ scrollToAMsg, msgHighlighted, goToCSM, moveToIMOptDisp, selMoveToIMOpt, \\ msgStoIsFull, cleanUpReqDisp, performCleanUp, msgMovedToIMDisp, accept.1\rangle$$

Further test scenarios that satisfy a given test purpose can be generated incrementally as explained in the previous section.

6 Constructing Sound Test Cases

In conformance testing, the minimum requirement for the generated test cases is that they do not reject correct implementations; they must be *sound*. In this section we show that our test case generation strategy always produces sound test cases.

CSP Input-Output Conformance To obtain soundness, conformance testing [20] firstly requires the definition of an implementation relation between the domain of specifications and the domain of implementations. In our work elements of such domains are expressed as CSP processes. Thus, to present our definition for such a relation we assume as *test hypothesis* [3] that there is a CSP process which specifies the implementation under test (IUT), say IUT_{CSP}.

We also assume that implementations satisfy the implementation protocol given by the definition below, which requires that an implementation always accept any input from its alphabet, and either recurses to accept another input or produces an output. In the following definition the notation $P \;|||\; Q$ represents the interleaving between the processes P and Q. In such a composition both processes communicate any event freely (no synchronisation).

Definition 2 *Let IUT_{CSP} be an implementation model, and consider the following process: $PROT = \Box\, i : \alpha_{IUT_{CSP}i} \bullet i \rightarrow (\Box\, o : \alpha_{IUT_{CSP}o} \bullet o \rightarrow PROT \;\Box\; PROT)$. IUT_{CSP} is a valid implementation model if the following holds.*

$$(IUT_{CSP} \;|||\; RUN(\alpha_{IUT_{CSP}o})) \;[\![\, \alpha_{IUT_{CSP}} \,]\!]\; PROT =_{\tau} PROT \tag{2}$$

In the above definition, equality in the traces model ($=_{\tau}$) stands for traces refinement in both directions. Therefore, the above equation can be verified using FDR. From now on we assume that any implementation model IUT_{CSP} obeys Definition 2.

Our implementation relation **cspio** (CSP Input-Output Conformance), formalised in Definition 3, is the basis for our generation of sound CSP test cases. Consider that $initials(P) = \{e \mid \langle e \rangle \in traces(P)\}$ yields the initial events offered by the process P, and the function $out(P, s)$ gives the set of output events of P after the trace s. More precisely, $out(P, s)$ = if $s \in traces(P)$ then $initials(P/s) \cap \alpha_{Po}$ else $\emptyset$. The relation **cspio** establishes that any output event observed in an implementation model IUT_{CSP} is also observed in the specification S, after any trace of S. In this case, IUT_{CSP} **cspio** S.

Definition 3 *Let IUT_{CSP} be an implementation model, and S a specification, such that $\alpha_{Sc} = \emptyset$, $\alpha_{Si} \subseteq \alpha_{IUT_{CSP}i}$, $\alpha_{So} \subseteq \alpha_{IUT_{CSP}o}$. Then,*

$$IUT_{CSP} \;\textbf{\textit{cspio}}\; S \Leftrightarrow \forall\, s : traces(S) \bullet out(IUT_{CSP}, s) \subseteq out(S, s)$$

The following theorem captures **cspio** using process refinement.

Theorem 1 *Let IUT_{CSP} be an implementation model, and S a specification, such that $\alpha_{Sc} = \emptyset$, $\alpha_{Si} \subseteq \alpha_{IUT_{CSP}i}$ and $\alpha_{So} \subseteq \alpha_{IUT_{CSP}o}$. The relation IUT_{CSP} **cspio** S holds iff the following refinement holds.*

$$S \sqsubseteq_{\tau} (S \;|||\; RUN(\alpha_{IUT_{CSP}o})) \;[\![\, \alpha_{IUT_{CSP}} \,]\!]\; IUT_{CSP} \tag{3}$$

Proof

$$S \sqsubseteq_\tau (S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))\,[|\,\alpha_{IUT_{CSP}}\,|]\,IUT_{CSP}$$

= [by the definition of $\sqsubseteq_\tau$]

$$traces(S) \supseteq traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))\,[|\,\alpha_{IUT_{CSP}}\,|]\,IUT_{CSP})$$

= [by assumption $\alpha_S \subseteq \alpha_{IUT_{CSP}}$ and definition $traces(P\,[|\,\alpha_P \cup \alpha_Q\,|]\,Q)$] in [15], page 54]

$$traces(S) \supseteq traces(S \,|||\, RUN(\alpha_{IUT_{CSP}\,o})) \cap traces(IUT_{CSP})$$

= [by set theory and predicate logics]

$$\begin{aligned}\forall\, s, x \bullet\ & s \frown \langle x\rangle \in (traces(S \,|||\, RUN(\alpha_{IUT_{CSP}\,o})) \cap traces(IUT_{CSP})) \Rightarrow \\ & s \frown \langle x\rangle \in traces(S)\end{aligned}$$

= [by definition $traces(P/s)$ in [15], page 48]

$$\begin{aligned}\forall\, s, x \bullet\ & \langle x\rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))/s) \cap traces(IUT_{CSP}/s) \Rightarrow \\ & \langle x\rangle \in traces(S/s)\end{aligned}$$

= [by set theory]

$$\begin{aligned}\forall\, s, x \bullet\ & \langle x\rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))/s) \wedge \\ & \langle x\rangle \in traces(IUT_{CSP}/s) \Rightarrow \\ & \langle x\rangle \in traces(S/s)\end{aligned}$$

= [by definition $initials(.)$ in [15], page 47]

$$\begin{aligned}\forall\, s, x \bullet\ & x \in initials((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))/s) \wedge \\ & x \in initials(IUT_{CSP}/s) \Rightarrow \\ & x \in initials(S/s)\end{aligned}$$

= [unfolding $initials(.)$]

$$\begin{aligned}\forall\, s, x \bullet\ & x \in \{\, e \mid \langle e\rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))/s)\} \wedge \\ & x \in \{\, e \mid \langle e\rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\ & x \in \{\, e \mid \langle e\rangle \in traces(S/s)\}\end{aligned}$$

= [by hypothesis $\alpha_{IUT_{CSP}} = \alpha_{IUT_{CSP}\,i} \cup \alpha_{IUT_{CSP}\,o}$]

$$\begin{aligned}\forall\, s, x \mid\ & x \in \alpha_{IUT_{CSP}\,i} \vee x \in \alpha_{IUT_{CSP}\,o} \bullet \\ & x \in \{\, e \mid \langle e\rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\,o}))/s)\} \wedge \\ & x \in \{\, e \mid \langle e\rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\ & x \in \{\, e \mid \langle e\rangle \in traces(S/s)\}\end{aligned}$$

= *[by predicate calculus]*

$$
\begin{array}{l}
\forall\, s, x \mid x \in \alpha_{IUT_{CSP}i} \bullet \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s)\} \wedge \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\} \\
\vee \\
\forall\, s, x \mid x \in \alpha_{IUT_{CSP}o} \bullet \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s)\} \wedge \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\}
\end{array}
$$

Assuming each proposition of disjunction in isolation we reach the same conclusion.

= *[assuming* $x \in \alpha_{IUT_{CSP}i}$*]*

$$
\begin{array}{l}
\forall\, s, x \mid x \in \alpha_{IUT_{CSP}i} \bullet \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s)\} \wedge \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\}
\end{array}
$$

= *[by predicate calculus]*

$$
\begin{array}{l}
\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s) \wedge \\
\qquad\qquad e \in \alpha_{IUT_{CSP}i}\} \wedge \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s) \wedge e \in \alpha_{IUT_{CSP}i}\} \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s) \wedge e \in \alpha_{IUT_{CSP}i}\}
\end{array}
$$

= *[by hypotheses* $\alpha_{Si} \subseteq \alpha_{IUT_{CSP}i}$ *and* $\alpha_{RUN(\alpha_{IUT_{CSP}o})i} \subseteq \alpha_{IUT_{CSP}i}$*]*

$$
\begin{array}{l}
\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s) \wedge \\
\qquad\qquad e \in \alpha_{(S|||RUN(\alpha_{IUT_{CSP}o}))i}\} \wedge \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s) \wedge e \in \alpha_{IUT_{CSP}i}\} \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s) \wedge e \in \alpha_{Si}\}
\end{array}
$$

= *[by hypothesis* $\alpha_{\Sigma i} \cap \alpha_{\Sigma o} = \emptyset$*]*

$$
\begin{array}{l}
\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \;|||\; RUN(\alpha_{IUT_{CSP}o}))/s)\} \cap \alpha_{(S|||RUN(\alpha_{IUT_{CSP}o}))o} = \emptyset \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \cap \alpha_{IUT_{CSP}o} = \emptyset \Rightarrow \\
\qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\} \cap \alpha_{So} = \emptyset
\end{array}
$$

= *[by definition of* $out(P, s)$*]*

$$
\begin{array}{l}
\forall\, s, x \bullet x \in out(S \;|||\; RUN(\alpha_{IUT_{CSP}o}), s) = \emptyset \wedge x \in out(IUT, s) = \emptyset \Rightarrow \\
\qquad\qquad x \in out(S, s) = \emptyset
\end{array}
$$

= *[by set theory]*

$$
\forall\, s : trace(S);\; x \bullet x \in out(IUT, s) \Rightarrow x \in out(S, s)
$$

o = *[by set theory]*

$$\forall\, s : traces(S) \bullet out(IUT, s) \subseteq out(S, s)$$

= *[by definition of* ***cspio****]*

$$IUT \ \ \textbf{cspio} \ \ S$$

The other branch.

= *[assuming* $x \in \alpha_{IUT_{CSP}\, o}$*]*

$$\begin{array}{l}\forall\, s, x \mid x \in \alpha_{IUT_{CSP}\, o} \bullet \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}))/s)\} \wedge \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \Rightarrow \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\}\end{array}$$

= *[by predicate calculus]*

$$\begin{array}{l}\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}))/s) \wedge \\ \qquad\qquad e \in \alpha_{IUT_{CSP}\, o}\} \wedge \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s) \wedge e \in \alpha_{IUT_{CSP}\, o}\} \Rightarrow \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s) \wedge e \in \alpha_{IUT_{CSP}\, o}\}\end{array}$$

= *[by hypotheses* $\alpha_{So} \subseteq \alpha_{IUT_{CSP}\, o}$ *and* $\alpha_{RUN(\alpha_{IUT_{CSP}\, o})o} \subseteq \alpha_{IUT_{CSP}\, o}$*]*

$$\begin{array}{l}\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}))/s) \wedge \\ \qquad\qquad e \in \alpha_{(S|||RUN(\alpha_{IUT_{CSP}\, o}))o}\} \wedge \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s) \wedge e \in \alpha_{IUT_{CSP}\, o}\} \Rightarrow \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s) \wedge e \in \alpha_{So}\}\end{array}$$

= *[by set theory]*

$$\begin{array}{l}\forall\, s, x \bullet x \in \{\, e \mid \langle e \rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}))/s)\} \cap \alpha_{(S|||RUN(\alpha_{IUT_{CSP}\, o}))o} \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(IUT_{CSP}/s)\} \cap \alpha_{IUT_{CSP}\, o} \Rightarrow \\ \qquad x \in \{\, e \mid \langle e \rangle \in traces(S/s)\} \cap \alpha_{So}\end{array}$$

= *[by definition of* $out(P, s)$*]*

$$\forall\, s, x \bullet x \in out(S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}), s) \wedge x \in out(IUT, s) \Rightarrow x \in out(S, s)$$

= *[by third proof step]*

$$\begin{array}{l}\forall\, s : trace(S);\ x \bullet x \in out(S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}), s) \wedge x \in out(IUT, s) \Rightarrow \\ \qquad\qquad x \in out(S, s)\end{array}$$

= *[by set theory]*

$$\begin{array}{l}\forall\, s : trace(S);\ x \bullet x \in (out(S \,|||\, RUN(\alpha_{IUT_{CSP}\, o}), s) \cap out(IUT, s)) \Rightarrow \\ \qquad\qquad x \in out(S, s)\end{array}$$

= *[by set theory]*

$\forall\, s : traces(S) \bullet out(S \,|||\, RUN(\alpha_{IUT_{CSP}o}), s) \cap out(IUT, s) \subseteq out(S, s)$

= *[Lemma 1]*

$\forall\, s : traces(S) \bullet \alpha_{IUT_{CSP}o} \cap out(IUT, s) \subseteq out(S, s)$

= *[by set theory]*

$\forall\, s : traces(S) \bullet out(IUT, s) \subseteq out(S, s)$

= *[by definition of **cspio**]*

IUT **cspio** S

$\diamond$

Lemma 1

$out((S \,|||\, RUN(\alpha_{IUT_{CSP}o})), s) = \alpha_{IUT_{CSP}o}$

Proof

$out((S \,|||\, RUN(\alpha_{IUT_{CSP}o})), s)$

= [by definition $out(.)$]

$initials((S \,|||\, RUN(\alpha_{IUT_{CSP}o}))/s) \cap (\alpha_{(S|||RUN(\alpha_{IUT_{CSP}o}))o})$

= [by processes alphabet's definition]

$initials((S \,|||\, RUN(\alpha_{IUT_{CSP}o}))/s) \cap (\alpha_{So} \cup \alpha_{RUN(\alpha_{IUT_{CSP}o})o})$

= [by definitions $RUN(s) = \Box\, ev : s \bullet ev \rightarrow RUN(s)$]

$initials((S \,|||\, RUN(\alpha_{IUT_{CSP}o}))/s) \cap (\alpha_{So} \cup \alpha_{IUT_{CSP}o})$

= [by assumption $\alpha_{So} \subseteq \alpha_{IUT_{CSP}o}$]

$initials((S \,|||\, RUN(\alpha_{IUT_{CSP}o}))/s) \cap \alpha_{IUT_{CSP}o}$

= [by definition $initials(.)$]

$\{e \mid \langle e \rangle \in traces((S \,|||\, RUN(\alpha_{IUT_{CSP}o}))/s)\} \cap \alpha_{IUT_{CSP}o}$

= [by definition $traces(P \,|||\, Q)$ in [15], page 67]

$\{e \mid \langle e \rangle \in \bigcup\{t \,|||\, u \mid t \in traces(S/s) \wedge u \in traces(RUN(\alpha_{IUT_{CSP}o})/s)\}\} \cap \alpha_{IUT_{CSP}o}$

= [by definitions $traces(RUN(s)) = (s)^*$ and P/s]

$$\{e \mid \langle e \rangle \in \bigcup\{t \mid\mid\mid u \mid t \in traces(S/s) \wedge u \in (\alpha_{IUT_{CSP}\,o})^*\}\} \cap \alpha_{IUT_{CSP}\,o}$$

= [by definition $s \mid\mid\mid t$ in [15], page 67]

$$\begin{array}{l}\{e \mid \langle e \rangle \in (\\ \quad \{\langle v \rangle \frown z \mid v \in initials(S/s) \wedge z \in (t \mid\mid\mid u) \wedge t \in traces(S/s/\langle v \rangle) \wedge \\ \qquad u \in (\alpha_{IUT_{CSP}\,o})^*\} \\ \quad \cup \\ \quad \{\langle v \rangle \frown z \mid v \in \alpha_{IUT_{CSP}\,o} \wedge z \in (t \mid\mid\mid u) \wedge t \in traces(S/s) \wedge \\ \qquad u \in (\alpha_{IUT_{CSP}\,o})^*\} \\ \quad) \\ \} \cap \alpha_{IUT_{CSP}\,o}\end{array}$$

= [by set theory]

$$\begin{array}{l}\{e \mid \langle e \rangle \in \{\langle v \rangle \frown z \mid v \in initials(S/s) \wedge z \in (t \mid\mid\mid u) \wedge t \in traces(S/s/\langle v \rangle) \wedge \\ \qquad u \in (\alpha_{IUT_{CSP}\,o})^*\}\} \cap \alpha_{IUT_{CSP}\,o} \\ \cup \\ \{e \mid \langle e \rangle \in \{\langle v \rangle \frown z \mid v \in \alpha_{IUT_{CSP}\,o} \wedge z \in (t \mid\mid\mid u) \wedge t \in traces(S/s) \wedge \\ \qquad u \in (\alpha_{IUT_{CSP}\,o})^*\}\} \cap \alpha_{IUT_{CSP}\,o}\end{array}$$

= [by set theory]

$$\begin{array}{l}\{e \mid \langle e \rangle \in \{\langle v \rangle \frown z \mid v \in initials(S/s) \wedge z \in (t \mid\mid\mid u) \wedge t \in traces(S/s/\langle v \rangle) \wedge \\ \qquad u \in (\alpha_{IUT_{CSP}\,o})^*\}\} \cap \alpha_{IUT_{CSP}\,o} \\ \cup \\ \alpha_{IUT_{CSP}\,o} \cap \alpha_{IUT_{CSP}\,o}\end{array}$$

= [from set theory]

$$\alpha_{IUT_{CSP}\,o}$$

$\diamond$

The intuition for this theorem is as follows. Consider an input event that occurs in IUT_{CSP}, but not in S. On the right-hand side of the refinement, the parallel composition cannot progress through this event, so it is refused. Because refused events are ignored in the traces model, new IUT_{CSP} inputs are allowed by the above refinement. The objective of the interleaving with the process $RUN(\alpha_{IUT_{CSP}\,o})$ is to avoid that the right-hand process refuses output events that the implementation can perform but S cannot. Thus, $RUN(\alpha_{IUT_{CSP}\,o})$ allows that such outputs be communicated to IUT_{CSP}. Finally, if IUT_{CSP} can perform such output events, then they appear in the traces of the right-hand side process, which falsifies the traces refinement.

In summary, the expression on the right-hand side captures new inputs performed by IUT_{CSP} generating deadlock from the trace where the input has occurred, in such a way that any event that comes after is allowed. Also, it keeps in the traces all the output events of IUT_{CSP} for the inputs from S, allowing a comparison in the traces models.

If we know IUT_{CSP} we can verify if IUT_{CSP} **cspio** S by checking (using FDR) the relation (3) directly. This is equivalent to generating all the traces of S and exercising them against the implementation according to **cspio**. However, in general we do not know IUT_{CSP} and the number of traces of S is infinite. Therefore, we need to exercise the implementation with a select subset of test cases and look for possible violations of IUT **cspio** S during the test execution.

Test Case and Successful Test Execution Following [20] we need to state what is the meaning of a test execution and the verdicts it can produce. The execution of a test TC against an implementation IUT_{CSP}, named $EX(IUT_{CSP}, TC)$, is the parallel composition $IUT_{CSP} \,|[\, \alpha_{IUT_{CSP}} \,]|\, TC$. Such an execution must yield a verdict $v \in VER = \{pass, fail, inc\}$. To check this in CSP, we need these verdict elements expressed as CSP processes. Thus, we use process $PASS = pass \rightarrow Stop$ to express when the test passes in the execution. Similarly $INC = inc \rightarrow Stop$ for an inconclusive execution, and $FAIL = fail \rightarrow Stop$ for a failed execution.

A test execution $EX(IUT_{CSP}, TC)$ for a given implementation IUT_{CSP} and a test case TC must always be successful. This is captured by the following definition.

Definition 4 *Let TC be a test case process, IUT_{CSP} an implementation model and $T = traces(EX(IUT_{CSP}, TC))$. The execution of TC against IUT_{CSP} is a successful test execution if the following holds.*

$$\forall\, t : T \mid (\neg\, \exists\, t' : T \mid t \neq t' \bullet t \leq t') \bullet last(t) \in \{pass, inc, fail\}$$

where $last(s)$ yields the last element of the sequence s.

The above definition states that the last element of each execution trace, which is not a prefix of any other execution trace, is a verdict event.

Constructing Sound Test Cases To construct a test case from a test scenario ts, first we create an *output complete* sequence lt that contains pairs $(ev_i, outs_i)$ such that ev_i is the i^{th} element of ts and $outs_i$ is the set of output events after the specification performs the trace $\langle ev_1, ..., ev_{i-1} \rangle$. Formally, $outs_i = out(S, \langle ev_i, ..., ev_{i-1} \rangle)$, for $1 < i \leq \#ts$, and $lt = \langle\, (get(1, ts), outs_1), ..., (get(\#ts, ts),\ outs_{\#ts}) \rangle$, where the expression $\#s$ yields the size of the sequence s.

The function $TC\_BUILDER(lt)$ defines how a sound test case can be constructed from a test scenario.

$$TC\_BUILDER(\langle\rangle) = PASS$$
$$TC\_BUILDER(\langle (ev_i,\ outs_i) \rangle \frown tail) = SUBTC((ev_i,\ outs_i));\ TC\_BUILDER(tail)$$

where

$$SUBTC((ev_i,\ outs_i)) = if\ (ev_i \in \alpha_{IUT_{CSP}i})\ then\ ev \rightarrow Skip$$
$$else\ (ev_i \rightarrow Skip \,\Box\, ANY(outs_i - \{ev_i\},\ INC) \,\Box$$
$$ANY(\alpha_{IUT_{CSP}o} - outs_i,\ FAIL))$$

Definition 5 *Let $ts \in traces(S)$ be a test scenario from S and lt an output complete sequence from ts. The traces of $TC\_BUILDER(lt)$ is*

$$\begin{aligned} &\forall\, s : prefixes(ts);\ o : \alpha_{IUT_{CSP}o} \mid s = \langle get(1, ts), \ldots, get(i, ts)\rangle \wedge 1 \leq i < \#lt \wedge \\ &\qquad\qquad s \frown \langle o\rangle \notin prefixes(ts) \bullet \\ &traces(TC\_BUILDER(lt)) = prefixes(ts \frown \langle pass\rangle) \cup \\ &\qquad\qquad \{\, s \frown \langle o, inc\rangle \mid o \in out_{i+1} \,\} \cup \\ &\qquad\qquad \{\, s \frown \langle o, fail\rangle \mid o \notin out_{i+1} \,\} \end{aligned}$$

where $prefixes(s)$ yields the set of prefixes for the sequence s. Formally,

$$\begin{aligned} &prefixes(\langle\rangle) = \{\langle\rangle\} \\ &prefixes(s \frown \langle last\rangle) = \{s \frown \langle last\rangle\} \cup prefixes(s) \end{aligned}$$

The process $TC\_BUILDER(lt)$ recursively applies the process $SUBTC$ for each pair $(ev_i, outs_i)$ of lt and yields the process $PASS$ when the last element of lt is reached. The goal of the process $SUBTC$ is to create the body of the test, inserting the verdicts fail and inconclusive at intermediate points of the test case according to the following.

If the event ev_i is an input, the test case communicates this event to the implementation, and finishes the verification of this test fragment successfully ($Skip$). Otherwise, if ev_i is an output, the test must be ready to synchronise with any output response of the IUT_{CSP} (output completeness), including ev_i. If IUT_{CSP} communicates ev_i, the test synchronises on this event and ends with success ($Skip$). Case the IUT_{CSP} communicates an event that belongs to $outs_i - \{ev_i\}$, the test reaches the verdict inconclusive since the IUT_{CSP} response is not exactly the one expected by the test scenario (ev_i), but it is a behaviour that is foreseen by the specification. Otherwise, if the IUT_{CSP} communicates an event that is not foreseen by the specification the test reaches the verdict fail.

Before we address soundness of a test case, the following theorem states that a test case constructed from $TC\_BUILDER$ terminates successfully when executed against an implementation model.

Theorem 2 *Let IUT_{CSP} be an implementation model, S a specification, ts a test scenario from S, such that $\alpha_{Sc} = \emptyset$, $\alpha_{Si} \subseteq \alpha_{IUT_{CSP}i}$ and $\alpha_{So} \subseteq \alpha_{IUT_{CSP}o}$. If lt is an output complete sequence of ts and $TC = TC\_BUILDER(lt)$, then the execution of TC against IUT_{CSP} is a successful test execution.*

Proof

Our goal is to reach the statement from Definition 4 assuming that TC is $TC\_BUILDER(lt)$.

$$T = traces(EX(IUT_{CSP}, TC\_BUILDER(lt)))$$

= [by definition $EX(.)$]

$$T = traces(IUT_{CSP} \,[\![\, IUT_{CSP} \,]\!]\, TC\_BUILDER(lt))$$

= *[by assumption* $\alpha_{TC\_BUILDER} \subseteq \alpha_{IUT_{CSP}} \cup VER$ *and definition* $P \ |[\ X \mid Y\ ]|\ Q$ *in [15], page 68]*

$$T = traces((IUT_{CSP} \ |[\ \alpha_{IUT_{CSP}} \mid \alpha_{TC\_BUILDER(lt)}\ ]|\ TC\_BUILDER(lt))$$

= *[by definition* $traces(P \ |[\ X \mid Y\ ]|\ Q)$ *in [15], page 60]*

$$\begin{aligned} T = \{ & t \in (\alpha_{IUT_{CSP}} \cup VER)^* \mid (t \upharpoonright \alpha_{IUT_{CSP}}) \in traces(IUT) \wedge \\ & (t \upharpoonright (\alpha_{IUT_{CSP}} \cup VER)) \in traces(TC\_BUILDER(lt))\} \end{aligned}$$

= *[by definition* $traces(TC\_BUILDER(lt))$*]*

$$\begin{aligned} T = \{ & t \in (\alpha_{IUT_{CSP}} \cup VER)^* \mid (t \upharpoonright \alpha_{IUT_{CSP}}) \in traces(IUT) \wedge \\ & (t \upharpoonright (\alpha_{IUT_{CSP}} \cup VER)) \in traces(TC\_BUILDER(lt)) \wedge \\ & \forall\, s : prefixes(ts);\ o : \alpha_{IUT_{CSP}\, o} \mid s = \langle get(1, ts), \ldots, get(i, ts)\rangle \wedge \\ & \qquad 1 \leq i < \#lt \wedge s \frown \langle o\rangle \notin prefixes(ts) \bullet \\ & \qquad traces(TC\_BUILDER(lt)) = prefixes(ts \frown \langle pass\rangle) \cup \\ & \qquad \{\, s \frown \langle o, inc\rangle \mid o \in out_{i+1}\,\} \cup \{\, s \frown \langle o, fail\rangle \mid o \notin out_{i+1}\,\} \\ \} & \end{aligned}$$

$\Rightarrow$ *[by hypothesis* $\alpha_{IUT_{CSP}} \cap VER = \emptyset$ *and Definition 2]*

$$\begin{aligned} \forall\, t, t' : T \bullet\ & t = t' \frown \langle m\rangle \in VER \wedge \{t, t'\} \subseteq (\alpha_{IUT_{CSP}} \cup VER)^* \wedge \\ & t \upharpoonright \alpha_{IUT_{CSP}} \in traces(IUT_{CSP}) \wedge \\ & t \upharpoonright (IUT_{CSP} \cup VER) \in traces(TC\_BUILDER(lt)) \end{aligned}$$

$\Rightarrow$ *[by predicate calculus]*

$$\forall\, t, t' : T \bullet t = t' \frown \langle m\rangle \in VER \Rightarrow last(t) \in VER$$

= *[by predicate calculus]*

$$\forall\, t : T \bullet (\forall\, t' : T \bullet t = t' \frown \langle m\rangle \in VER) \Rightarrow last(t) \in VER$$

$\Rightarrow$ *[by predicate calculus]*

$$\forall\, t : T \bullet (\forall\, t' : T \bullet t = t' \vee t \neq t' \wedge t > t') \Rightarrow last(t) \in VER$$

= *[by boolean algebra]*

$$\forall\, t : T \bullet (\forall\, t' : T \bullet (t = t \vee t \neq t') \wedge (t = t' \vee t > t')) \Rightarrow last(t) \in VER$$

= *[by predicate calculus]*

$$\forall\, t : T \bullet (\forall\, t' : T \bullet t = t' \vee t > t') \Rightarrow last(t) \in VER$$

= *[by predicate calculus]*

$$\forall\, t : T \bullet (\neg\, \exists\, t' : T \bullet \neg(t = t' \vee t > t')) \Rightarrow last(t) \in VER$$

= [by predicate calculus]

$$\forall\, t : T \bullet (\neg\, \exists\, t' : T \bullet t \neq t' \wedge t \leq t') \Rightarrow last(t) \in VER$$

= [by predicate calculus]

$$\forall\, t : T \mid (\neg\, \exists\, t' : T \mid t \neq t' \bullet t \leq t') \bullet last(t) \in VER$$

= [by definition VER]

$$\forall\, t : T \mid (\neg\, \exists\, t' : T \mid t \neq t' \bullet t \leq t') \bullet last(t) \in \{pass, inc, fail\}$$

$\diamond$

Soundness is stated as: if the test execution leads to a fail verdict then the implementation does not conform to the specification. A CSP test execution of a test TC with an implementation IUT_{CSP} fails when the test execution $EX(IUT_{CSP}, TC)$ has the event $fail$ as part of at least one of its traces.

Definition 6 *Let IUT_{CSP} be an implementation process, S the specification and TC a test case process. Then TC is a sound test case if the following holds.*

$$\langle fail \rangle \in traces(EX(IUT_{CSP}, TC) \setminus \alpha_{IUT_{CSP}}) \Rightarrow \neg(IUT_{CSP}\ \textbf{\textit{cspio}}\ S)$$

A CSP test suite is sound if all its tests are also sound. The following theorem states that a test case constructed from $TC\_BUILDER$ is sound.

Theorem 3 *Let S be a specification, ts a test scenario from S and IUT_{CSP} an implementation model, such that $\alpha_{Sc} = \emptyset$, $\alpha_{Si} \subseteq \alpha_{IUT_{CSP}i}$ and $\alpha_{So} \subseteq \alpha_{IUT_{CSP}o}$. If lt is an output complete sequence of ts, then the process $TC\_BUILDER(lt)$ is a sound test case.*

Proof

= [Assumption]

$$\langle fail \rangle \in traces(EX(IUT_{CSP}, TC\_BUILDER(lt)) \setminus \alpha_{IUT_{CSP}})$$

= [by definition of $EX(.)$]

$$\langle fail \rangle \in traces((IUT_{CSP} \,[\![\, \alpha_{IUT_{CSP}} \,]\!]\, TC\_BUILDER(lt)) \setminus \alpha_{IUT_{CSP}})$$

= [by assumption $\alpha_{TC\_BUILDER} \subseteq \alpha_{IUT_{CSP}} \cup VER$ and definition $P \;[\![\, X \mid Y \,]\!]\; Q$ in [15], page 68]

$$\langle fail \rangle \in traces((IUT_{CSP} \;[\![\, \alpha_{IUT_{CSP}} \mid \alpha_{TC\_BUILDER(lt)} \,]\!]\; TC\_BUILDER(lt)) \setminus \alpha_{IUT_{CSP}})$$

= [by definition $P \setminus X$ in [15], page 84; and from $\alpha_{TC\_BUILDER}$]

$$\langle fail \rangle \in traces((IUT_{CSP} \;[\![\, \alpha_{IUT_{CSP}} \mid \alpha_{IUT_{CSP}} \cup VER \,]\!]\; TC\_BUILDER(lt)) \upharpoonright (\alpha_{IUT_{CSP}} \cup VER) - \alpha_{IUT_{CSP}})$$

= *[by set theory]*

$$\langle fail \rangle \in traces((IUT_{CSP} \ [\![\alpha_{IUT_{CSP}} | \alpha_{IUT_{CSP}} \cup VER]\!] \ TC\_BUILDER(lt)) \upharpoonright VER)$$

= *[by definition* $traces(P \ [\![X \mid Y]\!] \ Q)$ *in [15], page 60]*

$$\begin{aligned}\langle fail \rangle \in \{ & s \in (\alpha_{IUT_{CSP}} \cup VER)^* \mid (s \upharpoonright \alpha_{IUT_{CSP}}) \in traces(IUT) \wedge \\ & (s \upharpoonright (\alpha_{IUT_{CSP}} \cup VER)) \in traces(TC\_BUILDER(lt))\} \upharpoonright VER\end{aligned}$$

⇒ *[by definition* $traces(TC\_BUILDER(lt))$*, and by hypothesis* $\alpha_{IUT_{CSP}} - VER = \emptyset$*]*

$$\begin{aligned}\exists\, s : prefixes(ts);\ & o \in \alpha_{IUT_{CSP}\,o} \mid s = \langle get(1, ts), \ldots, get(i, ts) \rangle \wedge 1 \leq i < \#lt \wedge \\ & s \frown \langle o \rangle \notin prefixes(ts) \bullet \\ & s \frown \langle o, fail \rangle \in traces(EX(IUT_{CSP}, TC\_BUILDER(lt))) \wedge \\ & s \frown \langle o, fail \rangle \in traces(TC\_BUILDER(lt)) \wedge s \frown \langle o \rangle \in traces(IUT_{CSP}) \wedge \\ & o \notin outs_{i+1}\end{aligned}$$

⇒ *[by definition* out_i*]*

$$\begin{aligned}\exists\, s : prefixes(ts);\ & o \in \alpha_{IUT_{CSP}\,o} \mid s = \langle get(1, ts), \ldots, get(i, ts) \rangle \wedge 1 \leq i < \#lt \wedge \\ & s \frown \langle o \rangle \notin prefixes(ts) \bullet \\ & s \frown \langle o, fail \rangle \in traces(EX(IUT_{CSP}, TC\_BUILDER(lt))) \wedge \\ & s \frown \langle o, fail \rangle \in traces(TC\_BUILDER(lt)) \wedge s \frown \langle o \rangle \in traces(IUT_{CSP}) \wedge \\ & o \notin out(S, s)\end{aligned}$$

⇒ *[by hypothesis* $ts \in traces(S)$ *and definition* $out(.)$*]*

$$\begin{aligned}\exists\, s : traces(S);\ & o \in \alpha_{IUT_{CSP}\,o} \mid s = \langle get(1, ts), \ldots, get(i, ts) \rangle \wedge 1 \leq i < \#lt \wedge \\ & s \frown \langle o \rangle \notin prefixes(ts) \bullet \\ & s \frown \langle o, fail \rangle \in traces(EX(IUT_{CSP}, TC\_BUILDER(lt))) \wedge \\ & s \frown \langle o, fail \rangle \in traces(TC\_BUILDER(lt)) \wedge s \frown \langle o \rangle \in traces(IUT_{CSP}) \wedge \\ & o \notin out(S, s) \wedge o \in out(IUT, s)\end{aligned}$$

⇒ *[by predicate calculus]*

$$\exists\, s : traces(S);\ o \in \alpha_{IUT_{CSP}\,o} \mid o \notin out(S, s) \wedge o \in out(IUT_{CSP}, s)$$

⇒ *[by predicate calculus and set theory]*

$$\exists\, s : traces(S) \bullet out(IUT_{CSP}, s) \not\subseteq out(S, s)$$

⇒ *[by predicate calculus and set theory]*

$$\neg(\forall\, s : traces(S) \bullet out(IUT_{CSP}, s) \subseteq out(S, s))$$

⇒ *[by definition* ***cspio****]*

$$\neg(IUT_{CSP} \ \textbf{cspio} \ S)$$

◇

To exemplify the construction of a sound test case, we assume the test scenario $UC_1\_ts_1$ and build the process $TC_1 = TC\_BUILDER\,(lt\_ts_1)$, where $lt\_ts_1 = \langle(goToMsgCenter, \emptyset), (IMFolderIsDisp, \{IMFolderIsDisp\}), (goToInbox, \emptyset), (inboxMsgsDisp, \{inboxMsgsDisp\}), (scrollToAMsg, \emptyset), (msgHighlighted, \{msgHighlighted\}), (goToCSM, \emptyset), (moveToIMOptDisp, \{moveToIMOptDisp\}), (selMoveToIMOpt, \emptyset), (msgMovedToIMDisp, \{msgMovedToIMDisp\})\rangle$. The resulting process is

$TC_1 = goToMsgCenter \rightarrow Skip;$
$(IMFolderIsDisp \rightarrow Skip \,\Box\, ANY(\alpha_{UC1o} - \{IMFolderIsDisp\}, FAIL));$
$goToInbox \rightarrow Skip;$
$(inboxMsgsDisp \rightarrow Skip \,\Box\, ANY(\alpha_{UC1o} - \{inboxMsgsDisp\}, FAIL));$
$scrollToAMsg \rightarrow Skip;$
$(msgHighlighted \rightarrow Skip \,\Box\, ANY(\alpha_{UC1o} - \{msgHighlighted\}, FAIL));$
$goToCSM \rightarrow Skip;$
$(moveToIMOptDisp \rightarrow Skip \,\Box\, ANY(\alpha_{UC1o} - \{moveToIMOptDisp\}, FAIL));$
$selMoveToIMOpt \rightarrow Skip;$
$(msgMovedToIMDisp \rightarrow PASS \,\Box\, ANY(\alpha_{UC1o} - \{moveToIMOptDisp\}, FAIL));$

According to Theorem 3 TC_1 is a sound test case.

7 The ATG

Abstract Test Generator (ATG) is a tool that we have developed to implement the test generation approach we propose. It constructs processes and refinements to be verified by FDR, as well as produces test cases. Fig. 5 displays the internal information workflow used by the proposed tool for test case generation. ATG receives as inputs a system specification process (SPEC) described in CSP, and a list of pseudo test purposes (PTP).

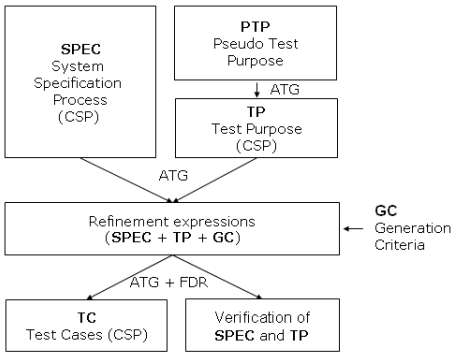

Fig. 5. Abstract Test Generator Tool

A pseudo test purpose (PTP) is a simpler representation of a test purposes. The aim of a PTP is to make the test purpose specification more user friendly, hiding from the

user the underlying notation of CSP. Fig. 6 shows the pseudo test purposes $PTP1$ that represents the test purposes $TP1$ from Section 5.

$$
\begin{aligned}
PTP1 = \; & any.\{y\} \rightarrow until.\{z\} \rightarrow accept.1 \\
& \mid any.\{z\} \rightarrow refuse.1 \\
& \mid not.\{y, z\} \rightarrow start
\end{aligned}
$$

Fig. 6. Pseudo test purpose

Each channel of a pseudo test purpose is mapped into a test purpose process, except the channel $start$ that is mapped into the test purpose process itself. Table 2 left-hand column shows the pseudo test purpose channels, and the right-hand column shows the test purpose process that represents the channel. The value v communicated by each channel is the value for the parameter $evset$ of the process on the right-hand column. The value for the parameter $next$ of the processes on the right-hand column is set automatically.

Table 2. Pseudo test purpose vs. test purpose

Pseudo Test Purpose Channel	Test Purpose Process
$accept.\{id\}$	$ACCEPT(id)$
$refuse.\{id\}$	$REFUSE(id)$
$any.v$	$ANY(...)$
$not.v$	$NOT(...)$
$match.v$	$MATCH(...)$
$matchs.v$	$MATCHS(...)$
$except.v$	$EXCEPT(...)$
$until.v$	$UNTIL(...)$

Furthermore, the tool receives the test generation criteria (GC). Since each refinement checking retrieves the shortest test scenario, the GC information is useful to set the number of test cases concerning the iterative process presented in Sect. 4.

Since ATG runs FDR in background, it can check classical properties of the specification like absence of deadlock, livelock and nondeterministic behaviour. Furthermore, it can verify the consistency of test purposes (if the scenario specified by the test purpose can be found in the specification). Also, ATG uses FDR Explorer [4] to retrieve from FDR the LTS representation of processes to produce a format for visualisation that can be displayed by a freeware graph visualisation tool [22].

ATG was implemented using the Java 1.5 technology that enables it to run over multiple platforms. Its server mode allows the user to run all the tool functionalities remotely, freeing the user from having to interact with FDR, while executing it from a desktop machine.

Case Study ATG has been applied to generate test cases in the context of mobile phone applications. We briefly describe one case study performed to compare a test suite generated using ATG (TSA — Test Suite from ATG) with a test suite manually designed (TSM — Test Suite Manually designed). Both test suites were generated for the same Motorola mobile application, and were compared with respect to test suite size, code coverage measured for test execution, bug detection rate and effort for design and execute the suites.

The selected application for the study was an SMS software used to compose, send and manage short messages in a specific Motorola mobile phone. The functional behaviour of the application was specified using the use case format of [2]. A CSP specification, the main input for ATG, was automatically extracted from the use case specification. Also, a pseudo test purpose was written for each software requirement to guide ATG test generation. Then, the CSP test suite was translated to the test format of [18], which is suitable for manual test execution. The effort to produce TSA was about 50% of the effort to produce TSM, considering the construction of the use case specification. The generation itself takes only a few seconds. Therefore, for generating additional test cases, the gain of the automated generation would be much more significant.

The TSM included 84 test cases, while the TSA included 37 generated test cases. This size difference is mainly related to the abstraction of the use case specification used to obtain the CSP specification. The use case model had a high abstraction level that yielded test cases in TSA with a higher level of abstraction than those in TSM. A unique test case from TSA could be compared to several test cases from TSM.

Both TSM and TSA test cases where executed against the same IUT. The effort to execute TSA was about 45% of the effort to execute TSM, and the bug detection rate was exactly the same in both cases, with the detection of two bugs each. The IUT software was instrumented to allow the code coverage tool to generate code coverage analysis reports. Considering the same coverage metrics, TSA and TSM had, in average, an equivalent coverage of the IUT.

8 Conclusions

The main contribution of this paper is a uniform strategy for generating sound test cases, based on the **cspio** conformance relation, from test scenarios extracted from CSP test models. All the elements of our approach are entirely characterised in terms of CSP processes and refinement notions. We have shown how to specify test models both for individual features and for feature interaction, from use case documents that are written in a controlled natural language (CNL). Test scenarios are incrementally generated from the test models as counter-examples of refinement verifications using the FDR tool; test selection is captured by CSP processes based on the concept of test purpose.

We have also built tool support to mechanise the entire approach. Figure 7 presents an overview of the ATG (Abstract Test Generator) tool workflow. The tool takes as input a test model (which is itself generated from use cases in CNL) and a set of test purposes. Internally, the tool generates a set of test scenarios that satisfy the test purposes, through refinement verifications, running FDR in background; the user can inform the number of scenarios to be generated. The test scenarios are then used to generate sound test

cases (still expressed as CSP processes). Finally, the test cases are translated back to CNL [18], yielding the test case suite.

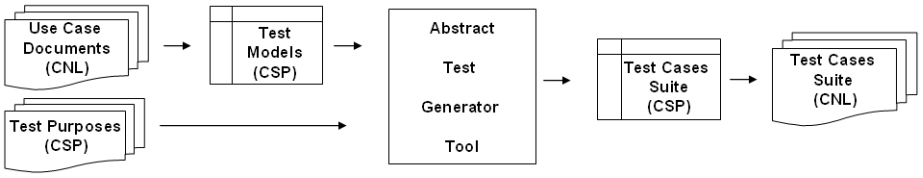

Fig. 7. Test automation workflow - The ATG tool

Tretmans [21, 20] outlines a formal testing theory and tool that is based on IOLTS (Input-Output LTS) models and on the implementation relation named **ioco**. Our relation **cspio** is similar to **ioco**; both use input and output events to define conformance. However **ioco** is formulated in terms of IOLTS, while **cspio** is defined in terms of the CSP denotational semantics. The relation **ioco** considers quiescence behaviours, that we currently forbid by checking that the implementation model obeys the protocol introduced in Definition 4; we plan to allow quiescence in a future work. Based on the **ioco** relation, Jard and Jéron [6] present the TGV tool that is able to select test cases based on test purposes and uses a test generation approach that is close to ours, but based on IOLTS. Expressing test models (particularly feature interaction) and test purposes in a process algebra has proved very convenient in our application domain.

Andrade et al. [1] made an alternative effort to capture the mobile devices application domain based on LTS. This has demanded a strategy to deal with individual features and a separate one to capture feature interaction. Our approach does not need explicit generation algorithms, and deals uniformly with features and (flexible patterns of) interactions.

Cavalcanti and Gaudel [3] stated the testability hypothesis for CSP and proposed a characterization of a test generation approach proved to be complete with respect to their implementation relation that is based on traces and failures refinement of CSP. However, they do not address test purposes; also, their work does not distinguish inputs and outputs, and does not propose an automatic approach to test generation.

Peleska and Siegel [13] present some implementation relations based on the semantic models of CSP. Their definitions are based on several refinement relations that define the observations of testing; however, unlike our approach, input and output are not observations. Scheneider [17] defines a partition that classifies refusable and non-refusable events, and high-level and low-level events, for the purposes of specifying fault-tolerance systems with CSP. He defines two conformance relations and refinement is used to check whether conformance holds, but no approach for test generation is proposed.

Based on our results on the formal composition of components and frameworks [14] we plan to explore compositional test generation, which avoids the retesting of already assembled components. We believe that this kind of application will emphasise

the distinguishing nature of our approach entirely based on a process algebra, where we can make explicit the application architecture, including the interaction patterns among components, unlike more operational models based on LTS or FSM.

Acknowledgements We would like to thank the feedback from the IFIP WG 2.3, from the UK Motorola Labs, and from the members of the CIn-BTC Research Project. Also we want to thank Lars Frantzen for feedbacks in an earlier draft, and Jim Woodcock for having suggested to us the relation (1).

References

1. Andrade W. et al. Interruption test case generation for mobile phone applications (in portuguese). In *XXV Brazilian Symposium in Computer Networks and Distributed Systems* (2007).
2. Cabral, G., and Sampaio, A. Formal Specification Generation from Requirement Documents. *Electron. Notes Theor. Comput. Sci. 195* (2008), 171–188. Best Paper Award.
3. Cavalcanti, A., and Gaudel, M.-C. Testing for refinement in csp. In *ICFEM* (2007), vol. 4789 of *LNCS*, Springer, pp. 151–170.
4. Freitas, L., and Woodcock, J. Fdr explorer. *ENTCS 187* (2007), 19–34.
5. Hierons, R. Checking states and transitions of a set of communicating finite state. *Microprocessors and Microsystems, Special Issue on Testing and testing techniques for real-time embedded software systems 24*, 9 (2001), 443– 452.
6. Jard, C., and Jéron, T. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf. 7*, 4 (2005), 297–315.
7. K. Bogdanov et al. Working together: Formal Methods and Testing. *ACM Computing Surveys* (Dec. 2003).
8. Ledru Y. et al. Test Purposes: Adapting the Notion of Specification to Testing. *ASE 00* (2001), 127.
9. Leitão D., Torres D., Barros F. A. Nlforspec: Translating natural language descriptions into formal test case specifications. In *SEKE* (2007), Knowledge Systems Institute Graduate School, pp. 129–134.
10. Formal Systems. *Failures-Divergence Refinement - FDR2 User Manual*. Formal Systems (Europe) Ltd, June 2005.
11. ISO 8807:1989. *LOTOS : A formal description technique based on the temporal ordering of observational behaviour*. ISO, 1989.
12. Milner, R. *Communication and Concurrency*. Prentice Hall, 1989.
13. Peleska, J., and Siegel, M. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19 (1997), 53–77.
14. Ramos, R., Sampaio, A., and Mota, A. Framework composition conformance via refinement checking. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing* (2008), vol. 23, pp. 119–125.
15. Roscoe, A. W., Hoare, C. A. R., and Bird, R. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.
16. Sampaio A. et al. Software test program: a software residency experience. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (2005), ACM Press, pp. 611–612.
17. Schneider, S. Abstraction and testing. In *FM '99, Wold Congress on Formal Methods-Volume I* (1999), Springer-Verlag, pp. 738–757.

18. Torres, D., ao, D. L., and de Almeida Barros, F. Motorola SpecNL: A Hybrid System to Generate NL Descriptions from Test Case Specifications. *HIS 0* (2006), 45.
19. Torres D., Leitão D., Barros F. A. Motorola SpecNL: A Hybrid System to Generate NL Descriptions from Test Case Specifications. *HIS 0* (2006), 45.
20. Tretmans, J. Testing concurrent systems: A formal approach. In *CONCUR'99* (1999), J. Baeten and S. Mauw, Eds., vol. 1664 of *LNCS*, Springer-Verlag, pp. 46–65.
21. Tretmans, J., and Belinfante, A. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review* (November 8–12 1999), pp. 8–12.
22. yWorks. yEd : Java Graph Editor. http://www.yworks.com/en/ products_yed_about.htm, Aug. 2006.