DOI 10.1007/s00165-012-0258-z
BCS © 2012
Formal Aspects of Computing (2014) 26: 441–490

Formal Aspects of Computing

Test generation from state based use case models

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

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

Abstract. We present a strategy for the automatic generation of test cases from parametrised use case templates that capture control flow, state, input and output. Our approach allows test scenario selection based on particular traces or states of the model. The templates are internally represented as CSP processes with explicit input and output alphabets, and test generation is expressed as counter-examples of refinement checking, mechanised using the FDR tool. Soundness is addressed through an input–output conformance relation formally defined in the CSP traces model. This purely process algebraic characterisation of testing has some potential advantages, mainly an easy automation of conformance verification and test case generation via model checking, without the need to develop any explicit algorithm.

Keywords: Natural language; Test model; Use cases; CSP; Test generation; Conformance testing

1. Introduction

Testing consists in verifying whether the actual system behaviour matches the intended one. Hence, testing is related to some model (sometimes a mental model) [UPL06], which is the basis for the construction of test cases. Test cases are then constructed to assess the correctness of particular features of the system. A good set of test cases is directly related to how adequately the model captures the features of the implementation under test (IUT). Nevertheless, designing test cases manually can yield inconsistent test cases even if the model is trustworthy. Moreover, when the model changes, test cases must be updated and this is not always feasible manually, mainly when the number of tests grows. So manual craft and execution of tests can be costly and error prone.

Correspondence and offprint requests to: S. Nogueira, E-mail: scn@cin.ufpe.br

The selection of a good set of test cases and their automation aim at making the testing process more effective, less susceptible to errors and less dependent on human interaction. The purpose of model-based testing (MBT) is to use explicit models to automatise testing. Instead of manual design, tests are generated by a tool that processes the input model. Complementary, the generated tests can be automatically run against an implementation. Conformance testing is a kind of MBT whose objective is to check whether an IUT satisfies its specification according to some defined relation (conformance relation); an inherent assumption is that the class of specifications can be modelled in some known formalism so that it can be related with the specification model (test hypothesis). There are several conformance relations [BJK+05] mainly based on formal notations like Finite State Machines (FSM) and Labelled Transition Systems (LTS). Based on a conformance relation, test cases can be automatically generated from the model using algorithms [HBB+09] that ensure the satisfaction of properties by the generated tests (e.g. soundness).

Despite the advances of conformance testing, both in theoretical and practical fields, there are process related barriers for its wider adoption, such as the introduction of new tools and paradigms in traditional testing flows. Forcing the users to adopt new tools and formal notations does not seem to work, so user-friendly notations and interactive tools are necessary to reduce the gap between formal specifications required by MBT tools and informal specifications adopted in standard testing processes, often described in natural language. A promising direction to overtake this barrier is to develop domain specific approaches [Ber07].

Due to its convenience and easy to use notation, and adherence to object-oriented development methodologies, use cases [Gro07] have been adopted as the input model for test generation in many development contexts. Despite being part of object-oriented development methodologies, use cases can be developed by analysts and testers who do not have object-oriented programming skills. Very often use cases are the only available requirements documentation. Moreover, from use cases it is possible to validate the system in the early stages of the development, minimising costs.

Particularly, use case templates [CS08] are the standard input models for conformance testing in the Brazil Test Center (BTC) project [Sam05], a cooperation between the Federal University of Pernambuco and Motorola Inc., in the context of testing embedded software that run in mobile phones. A use case template is a document that defines the syntactic elements for authoring use cases and their relationships. It is structured as a set of features, each one described as a set of use cases. A feature is a clustering of individual requirements that describe a cohesive, identifiable unit of functionality; often described as input and output events that flow sequentially between the actors (mainly the user) and the GUI of the IUT. Thus, use case templates are very suitable for the description of the features to be tested. In BTC, use cases are described in a domain specific language for mobile applications [Tor06, Lei07]; this is a Controlled Natural Language (CNL), which is a small subset of English with a fixed grammar. The template has proved suitable [NCT+07] to specify individual features (mobile device functionalities) as well as several patterns of feature interaction.

Originally, use cases did not have a formal semantics, required for a fully automated test generation and reasoning about the properties of the generated test cases. However, more recently, several semantics have been proposed to use cases [CANM08, NFLTJ06, BL02, WP99, HVFR05, SC08] aiming at generating test cases. LTS and FSM are the main models used as basis to automate test generation from use case models; these are very concrete models and often adopted as the operational semantics of more abstract process algebras like CSP [Ros98], CCS [Mil89] and LOTOS [88089]. Contrasting with operational models (such as LTS or FSM), process algebra models can naturally evolve to incorporate additional requirements; the operators of a process algebra also allow complex models to be built from simpler ones in a compositional way. Test generation can take advantage of this more abstract level to be formalised in terms of the process algebra itself, generating concise, precise and proved correct solutions.

In BTC, from the use cases, models expressed in the CSP process algebra [Ros98] are automatically constructed [CS08] and used as input for our CSP based automatic generation approach of test scenarios. In this approach, instead of developing explicit algorithms, test scenarios are obtained from counter-examples of refinement verifications using FDR [For05], a model checker for CSP. There is no explicit manipulation of state spaces or control flow; test selection is captured by CSP test purposes, which are CSP processes that describe the properties of interest to be captured by the generated tests. In this approach, soundness is established via a conformance relation, named **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. As test hypothesis it is assumed that the class of implementations to be tested can be specified by some I/O process. Generated test scenarios are used to construct test cases that are sound with respect to **cspio**, ensuring that, only incorrect implementations can fail the tests.

A broad range of features can be described through sequential control flows using the use case templates from [CS08], as long as data and state aspects are abstracted into events. Those templates do not allow the explicit specification of data and state. One advantage of such an abstraction is that the templates become simpler. Nevertheless, the price to be paid with this simplification is loosing preciseness about inputs, outputs and states. In order to be able to describe more elaborate flows, where, for instance, loops controlled by dynamic changes in the system state are present, we need more powerful use case templates. For testing, one particular advantage of specifying data is the possibility to select tests based on the current state of the specification. Moreover, data in the model can be used for the automatic generation of precise initial and final states of each test case. This is very useful for generating test cases automatically as well as to find the optimal test execution ordering to optimise test setup costs [LISA09].

In this paper, we both improve and extend an initial approach to test case generation based on the CSP process algebra [NSM08] by taking into account data (inputs, outputs, variables and parameters) in the description of features, as well as in the selection of test scenarios. As an extension of the template introduced in [CS08], we propose a parametrised and state based template to capture these new attributes. To keep compatibility with the original template, new features are included as fields that complement the CNL sentences without changing their structure. Aligned with UML diagrams, we also improve the original templates with inclusion and extension relations, which characterise more concisely certain links among use case flows. In our state based approach, conditions in general (and particularly of extension points) can be properly and concretely expressed using variables, whereas in the original template they would have to be abstracted using events. We also develop an automatic translation from the proposed templates to CSP. The CSP specification obtained from the use case templates are what we call I/O processes, formed of ordinary CSP processes together with explicit input and output alphabets. Such a specification combines processes that model control flows, with those that model state to record the current values of the use case variables; the latter include simple operations to read and update the state. Based on this representation, CSP test purposes can describe test scenarios that match particular states of the specification and can also be used to assess the system state after completing the test scenarios. An advantage of our approach, based on a process algebra, is modularity: the incremental generation of test scripts, possibly using selection directives, is essentially the same both for test models that include only control flow and those extended with state information. As already mentioned, this is possible because such scenarios result from counter-examples of refinement verifications, without the need to develop any explicit algorithm, unlike approaches based on more operational models like LTS, which involve the design of new algorithms for addressing model extensions [And07, CANM08]. Another contribution of this paper is a detailed proof of soundness of the derivation of test cases from test scenarios (Theorem 5.2); although the soundness theorem originally appeared in [NSM08], its proof has not been previously published. When detailing the proof of the refinement characterisation of **cspio** conformance verification (Theorem 5.1) we uncovered a subtle technicality which led to a small correction in the refinement expression, as detailed later on. Our overall strategy is currently implemented into two separate tools. The automatic translation from use case templates to CSP is implemented as a component of the TaRGeT framework. Details of a previous release of TaRGeT (with use case templates restricted to control flows) can be found in [FNSB10]; the extension to our state-based template is a contribution of the current work. Test case generation and selection from the CSP model is implemented in a tool called ATG [NSM11], whose extension to address state-based models is also a contribution of our current work. A current task is to incorporate ATG into the TaRGeT framework and produce a complete implementation of our overall state-based approach.

Several test generation approaches that input some form of use cases have been proposed, many based on graphical notations as UML [NS11, WP99, BL02, HVFR05], and some based on natural language descriptions [BG03, SC08, CS08]. However, none of them considers a natural language representation that mixes control and state representation, which can be used to select particular scenarios during test generation. Furthermore, none of these approaches considers formal properties of the generated tests.

An overview of our application domain (mobile device software) is introduced in Sect. 2. The CSP testing approach for control flow is presented in Sect. 3, which encompasses the translation from the use case templates to CSP models as well as the use of CSP refinement checking to generate and select test scenarios. This section also emphasises our proposal for capturing use case inclusion and extension relations in the templates. In Sect. 4 we present the state based use case templates, their CSP representation and the selection of tests that match particular system states. The theoretical basis is depicted in Sect. 5, which encompasses the CSP characterisation of conformance testing and shows how to obtain sound test cases from a set of test scenarios in terms of input–output CSP processes. Section 6 discusses related work, and Sect. 7 concludes and presents topics for ongoing and future research.

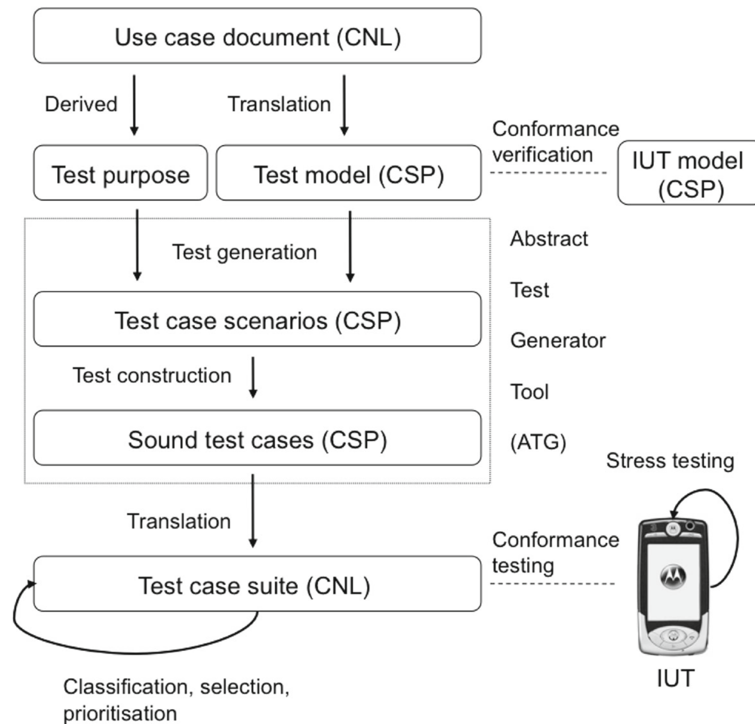

Fig. 1. Test automation workflow

2. Application domain

The development process of mobile phone software in the BTC project 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.

Figure 1 presents an overview of the automatic test generation approach in the BTC project. The main inputs are use case documents that describe the behaviour of the features to be tested, and selection criteria defined in terms of test purposes, derived from the use case document; the output is a test case suite appropriate for manual test execution. Another possibility is to generate test scripts and perform automatic test execution. The creation of scripts is supported by a tool [dPF08] that searches for the scripts stored in a database that can be reused. Input and output templates obey a Controlled Natural Language (CNL) standard [Tor06, Lei07] that can be translated to and from CSP. We have developed a tool, Abstract Test Generator (ATG), which plays a central role in the automation flow; ATG takes as input a CSP test model (which is generated from use cases in CNL [CS08]) and a set of test purposes. Internally, the tool generates a set of test scenarios that satisfy the test purposes; the user can inform the number of scenarios to be generated. The test scenarios are then used to construct sound test cases (still expressed as CSP processes). Finally, the test cases are translated back to CNL or translated into test scripts, yielding the test case suite. Concerning manual execution, the produced conformance tests are run by manually stimulating the GUI of an implementation under test (IUT), and comparing the obtained results with the expected ones to define a verdict. Auxiliary tools [MMYS09, LISA09] are used to automatically prioritise and select a subset of tests from a given test suite by identifying their relevance, and order them to optimize test execution effort. Complementing functional (conformance) testing with the purpose to find crashes in the software, test scripts (Java) are used to automatically perform stress testing against the GUI of an IUT [BMAF10]. Still in this setting, but considering an alternative to testing, it is possible to automatically verify the conformance of (a model of) the IUT against the test model. This is achieved by comparing the IUT model with the test model using the FDR tool [SNM09].

UC1 - Moving to Important Messages Folder

Main Flow

From Step: START
To Step: END

Step Id	Action	System State	System Response
1M	Go to folder.		All Messages are displayed.
2M	Select message(s).		Message(s) are highlighted.
3M	Select "Move to Important Messages" option.		"Message moved to Important Messages folder" is displayed.

(a) Main flow

From Step: 2M
To Step: START

Step Id	Action	System State	System Response
1A	Select "Move to Important Messages" option.		"Message storage has not enough space" is displayed. "Clean Up request" is displayed.

From Step: 1A
To Step: START

Step Id	Action	System State	System Response
1M	Select important message(s).		Message(s) are highlighted.
2M	Clean up selected message(s).		Clean is performed.

(b) Alternative flows

Fig. 2. Important messages mobile feature (F1)

In what follows, we overview the use case documents introduced in [CS08], based on Fig. 2a, b.

Use Cases Documents A use case document is formed of a set of features, each one described as a set of use cases. A 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 can be referenced (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 a user input action (User Action column), the expected system output in the System Response column, and the condition required to enable the input action (System State column). It is not mandatory to define both action and response for a step, but at least one of them must be specified; the condition is optional.

Figure 2a shows the main flow of the use case F1_UC1 of the Important Messages Feature, named F1. Such a flow specifies the sequence of actions that the user must perform to move a message from a folder to the Important Messages folder: go to a folder, select messages and choose the option to move. After moving a message, a confirmation is displayed and the flow finalises.

The fields `From Step` and `To Step` are used to indicate the set of steps from where the flow must start and continue, respectively. 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 2b (top) shows a possible alternative flow for the Important Messages use case. It specifies that after step 2M of the main flow the action to move selected messages does not succeed: a message notifies the situation that a clean up is requested; and the flow continues in the first step of the main flow. Since the steps 3M and 1A can perform a common action at the same time, regardless of system condition (move selected messages), the system response for such an action is nondeterministically chosen by the system. Still in Fig. 2b (bottom) a different alternative flow specifies the possibility, after step 1A, to delete messages in the Important Messages folder, and continues in the main flow. The exception flows are similar to alternative flows, except for representing exceptional behaviours.

3. Testing for control

In this section we introduce our approach to test case generation from CSP models, originally presented in [NSM08]. This approach is restricted to models with control flow. In the next section we extend this approach to consider parametrised and state based models, as well as inclusion and extension operations on use case templates. Its main feature is that test generation and selection are formulated as simple refinement verifications using the FDR refinement checker.

3.1. Test models as CSP processes

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

Although CSP is a very expressive process algebra, and, therefore, convenient to express our theory, there is no semantic distinction between input and output events in CSP. A model in our theory is a tuple $M = (P, A_I, A_O)$, where P is an ordinary CSP process representing the model behaviour, A_I the set of input events, and A_O the set of output events, with $A_I \cap A_O = \emptyset$ and $\alpha_P \subseteq A_I \cup A_O \cup \{\checkmark\}$; this model is called an I/O process.

Basic CSP operators as prefix, external (internal) choice and sequential composition are suitable to model the control flow of 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. A channel is an important abstraction in CSP for a set of events with common prefix. Let T be a type and c a channel that can communicate a value of type T (declared as $c : T$). Such a channel declaration represents the set of events $\{c.x \mid x \in T\}$, which can be more concisely denoted by the expression $\{\mid c \mid\}$. The external choice operator $P = Q \,\square\, R$ indicates that the process P can behave as Q or R; the choice is made by the environment. The internal choice operator $P = Q \sqcap R$ indicates that P can behave as Q or R; the choice is made non-deterministically by the process. The sequential composition $P; \ Q$ behaves like P until it terminates successfully, when the control passes to Q. Additionally, consider the CSP expression $P = let \ \ldots \ within \ Q$ that behaves as the process Q, whose context is augmented by the local definitions enclosed in the $let \ldots within$ clause. The guarded process $b \ \& \ P$ behaves as P if b evaluates to true, otherwise as $Stop$; formally it is equivalent to the expression $if \ b \ then \ P \ else \ Stop$.

Parallel composition is a very powerful CSP operator. Consider the process $P \ |[X]| \ Q$ that stands for the generalised parallel composition of the processes P and Q with synchronisation set X. This expression states that P and Q must synchronise on events that belong to X. Each process can evolve independently for events that are not in X. Such a composition terminates successfully if, and only if, the left and the right-hand side processes do terminate. Consider that 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), thus being equivalent to $P \ |[\{\}]| \ Q$.

Consider the replicated construction of CSP $\bigoplus x : A \bullet F(x)$, where $\bigoplus$ is a choice or parallel composition operator, x is a value from set A, and $F(x)$ any process expression involving x. This construction behaves as the process $F(x_1) \bigoplus \cdots \bigoplus F(x_k)$, for $A = \{x_1, \ldots, x_k\}$. For instance, the expression for the replicated external choice $\square \, x : A \bullet F(x)$ is equivalent to $F(x_1) \,\square \cdots \square\, F(x_k)$.

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

Let $R = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a renaming relation whose domain coincides with the alphabet of the process P. Thus, the CSP process $P[\![R]\!]$ represents P renamed according to R, that is, $P[\![R]\!]$ communicates the events of $\{y \mid x \ R \ y\}$ whenever P communicates x, provided $x \in dom(R)$. Events not in $dom(R)$ are unchanged in $P[\![R]\!]$. For instance, let $R1 = \{(a, a), (a, b)\}$ and $P1 = a \rightarrow Stop$, so $P1[\![R1]\!] = a \rightarrow Stop \,\square\, b \rightarrow Stop$. $P[\![R]\!]$ can be alternatively represented as $P[\![^{x_1, \ldots, x_N}/_{y_1, \ldots, y_N}]\!]$. It is worth noting that renaming can create new behaviour as our previous example has demonstrated.

Finally, consider the process $RUN(s) = \square \, ev : s \bullet ev \rightarrow RUN(s)$ that continuously offers the events from the set s, and $P \bigtriangleup Q$ which indicates that Q can interrupt the behaviour of P if an event offered by Q is communicated.

As an example of the CSP notation, consider the following specification.

$$
\begin{array}{l}
F1_UC1 = let \\
\quad F1_UC1_START = F1_UC1_1M \\
\quad F1_UC1_1M = goTo \rightarrow msgsDisp \rightarrow Skip;\ (F1_UC1_2M) \\
\quad F1_UC1_2M = selectMsgs \rightarrow msgsHighlighted \rightarrow Skip;\ (F1_UC1_3M \Box F1_UC1_1A) \\
\quad F1_UC1_3M = selMoveToIMOpt \rightarrow msgMovedToIMDisp \rightarrow Skip \\
\quad F1_UC1_1A = selMoveToIMOpt \rightarrow cleanUpReqDisp \rightarrow Skip;\ (F1_UC1_START \Box F1_UC1_1B) \\
\quad F1_UC1_1B = selectImpMsgs \rightarrow msgsHighlighted_ \rightarrow Skip;\ F1_UC1_2B \\
\quad F1_UC1_2B = cleanUpMsgs \rightarrow cleanUpPerformed \rightarrow Skip;\ F1_UC1_START \\
within\ F1_UC1_START
\end{array}
$$

This process is obtained by applying the translation approach [CS08] to the main and alternative flows of the use case of the Important Messages Feature (Fig. 2). For conciseness, we abbreviate the event names that represent the elements of the use case templates. The CSP process that models a use case, like $F1_UC1$ above, is defined in terms of several auxiliary processes, one for each step flow, whose name is suffixed with the respective step id, including $START$. The process $F1_UC1$ behaves like the process $F1_UC1_START$, which itself invokes the process that captures the first step flow, and so on. In general, the process that models the start of the flow is a choice between the first steps of the flows that start from it (references in the `From step` field).

The CSP process for each of the other steps is modelled as a sequence of CSP events encoding the step action and response, respectively, which prefixes the successful termination of the step (*Skip*). Analogously to the start step, if the step continues through other steps it is sequentially composed with its continuation. For instance, the process $F1_UC1_1M$ models the sequence of action and response of the step 1M (Fig. 2a), which is composed with the subsequent step of the flow ($F1_UC1_2M$). Continuing the flow, the process $F1_UC1_2M$ models the behaviour of step 2M whose continuation is the choice between the subsequent step flow (3M) and step 1A of the first alternative flow (Fig. 2b): $F1_UC1_3M \Box F1_UC1_1A$. The step 3M does not have a continuation, thus it ends with successful termination.

The process $F1_UC1_1A$ models the step 1A of the first alternative flow. Its continuation is the choice between the start process and the step 1B of the second alternative flow ($F1_UC1_START \Box F1_UC1_1B$). Finally, the processes $F1_UC1_1B$ and $F1_UC1_2B$ model the behaviour of the steps 1B and 2B of the second alternative flow. The former process leads to the second, and the continuation of the second is the start process of the main flow.

Finally, the I/O process of the Important Messages feature use case is the triple $F = (F1_UC1, A_{I_F}, A_{O_F})$, where the alphabet sets A_{I_F} and A_{O_F} contain the input and output events, respectively, and are defined as follows.

$$A_{I_F} = \{goTo, selectMsgs, selMoveToIMOpt, selectImpMsgs, cleanUpMsgs\}$$

$$
\begin{array}{ll}
A_{O_F} = \{ & msgsDisp, msgsHighlighted, msgMovedToIMDisp, cleanUpReqDisp, msgsHighlighted_, \\
& cleanUpPerformed\}
\end{array}
$$

Semantic Models for CSP Trace semantics is the simplest model for a CSP process. Our test generation approach is solely based on this model. The traces of a process P, given by $\mathcal{T}(P)$, correspond to the set of all possible sequences of events P can communicate.

Consider that P and Q are two CSP processes, and that the event $\checkmark$ does not belong to Σ. Also, $s_1 \frown s_2$ stands for the concatenation of sequences s_1 and s_2, $\langle e \rangle$ is a sequence containing the element e and $\#s$ yields the size of the sequence s. The notation $s \setminus X$ yields the subsequence of s resulting from removing the events from the set X.

Definition 1 presents the traces for CSP primitive processes as well as for the CSP operators used in this work. A complete definition for all CSP operators can be found in [Ros98].

Definition 1 (Trace semantics of processes). Let P and Q be CSP processes and Σ the set of all specified events.

$$
\begin{array}{lcl}
\mathcal{T}(Skip) & = & \{\langle\rangle, \langle\checkmark\rangle\} \\
\mathcal{T}(Stop) & = & \{\langle\rangle\} \\
\mathcal{T}(a \rightarrow P) & = & \{\langle\rangle\} \cup \{\langle a\rangle \frown s \mid s \in \mathcal{T}(P)\} \\
\mathcal{T}(P \,\Box\, Q) & = & \mathcal{T}(P) \cup \mathcal{T}(Q) \\
\mathcal{T}(P \sqcap Q) & = & \mathcal{T}(P \,\Box\, Q) \\
\mathcal{T}(P;\ Q) & = & (\mathcal{T}(P) \cap \Sigma^*) \cup \{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathcal{T}(P) \wedge t \in \mathcal{T}(Q)\} \\
\mathcal{T}(P \setminus X) & = & \{s \upharpoonright \Sigma - X \mid s \in \mathcal{T}(P)\} \\
\mathcal{T}(P/s) & = & \{t \mid s \frown t \in \mathcal{T}(P)\} \\
\mathcal{T}(P \triangle Q) & = & \mathcal{T}(P) \cup \{s \frown t \mid s \in \mathcal{T}(P) \cap \Sigma^* \wedge t \in \mathcal{T}(Q)\} \\
\mathcal{T}(P \mid[X]\mid Q) & = & \bigcup\{s \mid[X]\mid t \mid s \in \mathcal{T}(P) \wedge t \in \mathcal{T}(Q)\}
\end{array}
$$

All processes include the empty trace ($\langle\rangle$). The $Skip$ process produces the event $\checkmark$ to indicate successful termination, and $Stop$ communicates no visible events. All non-empty traces of $a \rightarrow P$ are prefixed by a. Internal and external choices are not distinguished in the traces model. Both result in the union of the traces of the two operands. The traces of the sequential composition of two processes are the ones of the first process, but removing $\checkmark$ by $(\mathcal{T}(P) \cap \Sigma^*)$, and those formed of the concatenation of these traces with the ones produced by the second process. The traces resulting from hiding a set of events is given by preserving only those events that are not in X $(s \upharpoonright \Sigma - X)$, where the notation $t \upharpoonright A$ stands for the restriction of the sequence t to the elements of the set A. If $s \frown t$ is a trace of P, then t is a trace of P/s. The traces of $P \triangle Q$ are those of P plus non-termination traces of P ($\checkmark \notin \Sigma$), augmented with the traces of Q. The semantics of parallel composition uses an operator on traces $(s \mid[X]\mid t)$ which takes into account the synchronisation traces and all possible forms of interleavings between the traces of the two processes operating in parallel. The behaviour of the parallel composition is that the events from the synchronisation set must evolve together; other events can evolve independently. Below is the formal definition for the parallel composition of traces that is used in the definition of parallel composition of processes. Consider that $s, t \in \Sigma^*$, x is a member of the set X, and y is a member of $\Sigma - X$.

$$
\begin{array}{ll}
s \mid[X]\mid t & = t \mid[X]\mid s \\
\langle\rangle \mid[X]\mid \langle\rangle & = \{\langle\rangle\} \\
\langle\rangle \mid[X]\mid \langle x\rangle & = \emptyset \\
\langle\rangle \mid[X]\mid \langle y\rangle & = \{\langle y\rangle\} \\
\langle x\rangle \frown s \mid[X]\mid \langle y\rangle \frown t & = \{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown s \mid[X]\mid t\} \\
\langle x\rangle \frown s \mid[X]\mid \langle x\rangle \frown t & = \{\langle x\rangle \frown u \mid u \in s \mid[X]\mid t\} \\
\langle x\rangle \frown s \mid[X]\mid \langle x'\rangle \frown t & = \emptyset \text{ if } x \neq x' \\
\langle y\rangle \frown s \mid[X]\mid \langle y'\rangle \frown t & = \{\langle y\rangle \frown u \mid u \in s \mid[X]\mid \langle y'\rangle \frown t\} \\
 & \quad \cup \{\langle y'\rangle \frown u \mid u \in \langle y\rangle \frown s \mid[X]\mid t\}
\end{array}
$$

For instance, according to the definitions above $\mathcal{T}(RUN(A))$ equals A^*, which is the set of sequences formed of events from A, including the empty sequence.

It is possible to compare the traces semantics of two processes by refinement verification using one of the CSP refinement checking tools, such as FDR [For05], PAT [SLD08] or ARC [PY96]. In this work we use FDR because of its maturity and its notation CSP$_M$ [Sca98], a machine-readable dialect of CSP that was developed as the input language for FDR and the CSP Process Explorer [For11]. CSP$_M$ combines the CSP process algebra with a functional language.

A process Q refines a process P in the traces model, say $P \sqsubseteq_\tau Q$, if and only if $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$. If the refinement does not hold, FDR yields a trace (the shortest counter-example), say ce, such that $ce \in \mathcal{T}(Q)$ but $ce \notin \mathcal{T}(P)$. For instance, $F1_UC1_3M \,\Box\, F1_UC1_1A \sqsubseteq_\tau F1_UC1_1A$ holds, since $\mathcal{T}(F1_UC1_1A)$ is a subset of $\mathcal{T}(F1_UC1_3M \,\Box\, F1_UC1_1A)$. However, the relation $Skip \sqsubseteq_\tau accept.1 \rightarrow Stop$ does not, since $\langle accept.1\rangle \in \mathcal{T}(accept.1 \rightarrow Stop)$ but $\langle accept.1\rangle \notin \mathcal{T}(Skip)$. Thus, the trace $\langle accept.1\rangle$ is a counter-example for the last refinement expression above.

Other more elaborate semantic models of CSP are the failures and the failures-divergences models. The former captures nondeterminism and deadlock situations, whereas the latter captures livelocks as well. See [Ros98] for further details.

3.2. Test scenario generation

Given a test model S and a safety property Φ, we can obtain the traces of S that satisfy Φ (for example, traces from S that lead to a successful termination). 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 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. Then, S' is defined in such a way that for all test scenarios $ts \in \mathcal{T}(S)$ that satisfy Φ, there is a trace $ts ^\frown \langle m \rangle \in \mathcal{T}(S')$, such that $m \in MARK$ and $MARK \cap \alpha_S = \emptyset$. As a consequence $ts ^\frown \langle m \rangle \notin \mathcal{T}(S)$, 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, say ts_1, is retrieved by FDR when $S \sqsubseteq_{\tau} S'$ does not hold.

Let $\Phi(ts)$ be a predicate that evaluates to true iff $ts \in \mathcal{T}(S)$ satisfies Φ. Then, the expression below formalises the relation between the traces of S and those of S'.

$$\forall\, ts \mid \Phi(ts) \bullet \exists\, m : MARK \bullet ts ^\frown \langle m \rangle \in \mathcal{T}(S') - \mathcal{T}(S)$$

The difference between the traces of S' and those of S is formed of test scenarios satisfying Φ extended with a suffix $\langle m \rangle$. In other words, if $\Phi(ts)$ then $ts ^\frown \langle m \rangle$ belongs to the set of counter-examples of $S \sqsubseteq_{\tau} S'$, and vice-versa.

To illustrate the proposed approach, we show how to generate a set of test scenarios ($ts \in \mathcal{T}(S)$), which lead the test model to successful termination. Consider the process $ACCEPT(id) = accept.id \to Stop$ 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 the relation ($S \sqsubseteq_{\tau} S'$) yields as counter-examples test scenarios that lead the specification to successful termination (if they exist).

For example, checking the relation $F1_UC1 \sqsubseteq_{\tau} F1_UC1$; $ACCEPT(1)$ using FDR results in the shortest counter-example, as displayed below.

$$F1_UC1_ts_1 = \langle goTo, msgsDisp, selectMsgs, msgsHighlighted, selMoveToIMOpt, msgMovedToIMDisp, accept.1 \rangle$$

The above trace (ignoring the mark event $accept.1$) is the shortest successful termination test scenario to $F1_UC1$. It corresponds to the main use case flow of the Important Messages Feature (Fig. 2a).

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. Formally, $Proc(s) = if\,(s \neq \langle\rangle)\ then\ head(s) \to Proc(tail(s))\ else\ Stop$, such that $head(s)$ and $tail(s)$ are functions that yield the head and the tail of a non-empty sequence s, respectively. For instance, $Proc(\langle a, b, c \rangle)$ yields the process $a \to b \to c \to Stop$. 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 ✓.

The second counter-example is generated from S using the previous refinement, but with the process formed by the counter-example ts_1 ($Proc(ts_1)$) as an alternative to S on the left-hand side: $S \,\square\, Proc(ts_1) \sqsubseteq_{\tau} S'$. As $\mathcal{T}(S \,\square\, Proc(ts_1))$ is equivalent to $\mathcal{T}(S) \cup \{ts_1\}$, 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.

$$S \,\square\, Proc(ts_1) \,\square\, Proc(ts_2) \,\square\, \cdots \square Proc(ts_n) \sqsubseteq_{\tau} S' \qquad (1)$$

Continuing the selection of successful termination traces of $F1_UC1$, checking the relation $F1_UC1 \square Proc$ $(F1_UC1_ts_1) \sqsubseteq_{\tau} F1_UC1$; $ACCEPT(1)$ yields a second counter-example.

$$F1_UC1_ts_2 = \langle goTo, msgsDisp, selectMsgs, msgsHighlighted, selMoveToIMOpt, cleanUpReqDisp, goTo, msgsDisp, selectMsgs, msgsHighlighted, selMoveToIMOpt, msgMovedToIMDisp, accept.1 \rangle$$

The above trace is another successful termination test scenario for $F1_UC1$. It corresponds to the behaviour of the first alternative flow of the Important Messages feature (Fig. 2b (top)) followed by the main flow. Since the CSP model for the feature has infinite traces leading to successful termination, we can always increase the expression to generate lengthier test scenarios.

In a test generation approach based on refinement counter-examples, it is not possible to directly measure the coverage of the specification structure. Remarkably, if each input and output event of a use case is uniquely identified in the CSP model, the coverage of the use case steps can be measured by looking into the events of the generated test scenarios. However, the number of states and transitions that a set of counter-examples covers can be measured if we consider the transition set of the operational model for the specification process (LTS).

3.3. Test scenario selection

Although successful termination can itself be used as a selection criteria, as illustrated in the previous section, this section shows a more flexible strategy for selecting a set of test scenarios from a test model S based on the concept of a test purpose TP, also described as a CSP process. A CSP test purpose is based on the notion introduced in [LdBB+01]: a test purpose is a partial specification describing the characteristics of the desired tests. More concretely, it specifies the traces (safety property) that the generated test scenarios must have. The definition below formalises the concept.

Definition 2 (CSP Test Purpose). Let TP and S be CSP processes, m an event from $MARK$, and $X \subseteq (\alpha_S)^*$ a subset of the traces constructed from α_S. The process TP is a test purpose for S if it is deterministic and $\forall\, t : \mathcal{T}(TP) \bullet (t \in X) \vee (t \notin X \wedge t = t' \frown \langle m \rangle \wedge t' \in X)$.

The traces of TP belong to X or end with a mark event: removed the mark event the resulting trace belongs to X.

To ease the task of writing TP in CSP following Definition 2, we provide a set of primitive processes that can be combined to design possibly more elaborate test purposes.

The primitive $ANY(evset : \mathbb{P}\,\alpha_S, next) = \Box\, ev \;:\; evset \;\bullet\; ev \rightarrow next$ performs basic selection. It selects the events offered by the specification that belong to $evset$. If any of these events is communicated, it behaves as $next$. Otherwise, it deadlocks.

The process $UNTIL(\alpha_S, evset : \mathbb{P}\,\alpha_S, next) = RUN(\alpha_S - evset) \triangle ANY(evset, next)$ selects all sequences offered by the specification events until it engages in some event that belongs to $evset$. In [NSM11] one can find a comprehensive list of primitives.

The following is an example of a test purpose TP_1 that is used to select scenarios from $F1_UC1$. The objective of TP_1 is to select from $F1_UC1$ test scenarios whose final output is a message confirming that the selected important message is moved to the folder ($msgMovedToIMDisp$); this must happen after the user has performed a cleanup action ($cleanUpMsgs$).

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

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

Based on the test scenario generation approach from the previous subsection, 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 matches a test scenario, when TP communicates an event $mark \in MARK$. 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 is a test scenario. 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.

The parallel product of the test purpose TP and the I/O process $M = (P, A_I, A_O)$ is $PP(M, TP) = P \;|[A_I \cup A_O]|\; TP$. The refinement relation that yields test scenarios is $P \sqsubseteq_\tau PP(M, TP)$. Considering again our

example, the shortest test scenario from F that matches TP_1 is obtained from a counter-example of the relation $F1_UC1 \sqsubseteq_{\tau} PP(F, TP_1)$, where $PP(F, TP_1) = F1_UC1 \,|[A_{I_F} \cup A_{O_F}]|\, TP_1$. The counter-example is given below.

$$F1_UC1_TP_1_ts_1 = \langle goTo, msgsDisp, selectMsgs, msgsHighlighted, selMoveToIMOpt, cleanUpReqDisp, selectImpMsgs, msgsHighlighted, cleanUpMsgs, cleanUpPerformed, goTo, msgsDisp, selectMsgs, msgsHighlighted, selMoveToIMOpt, msgMovedToIMDisp, accept\rangle$$

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

4. Testing for data

The testing selection approach introduced in the previous section is restricted to control flow. No notion of state is taken into account. As a major contribution of this paper, use case templates and test case generation and selection are extended with an explicit notion of state. Use cases are extended with constructions for state update, input, output and parameterisation. Moreover, documents become more structured, by allowing use cases to be related through inclusion and extension mechanisms from UML 2.0 [Gro07]. The document structure is compositionally translated to CSP processes. The CSP specification combines processes that model control flows with those that model state to record the current values of the use case variables; the latter include special events to read and update the state. Based on such a structured specification, CSP test purposes can describe test scenarios that match particular states of the specification. The test selection approach described in the previous chapter is extended to consider the current values of use case variables.

4.1. State based use cases

Before introducing the new use case template, we present a refactoring of the Important Messages Feature. Such a refactoring is the running example of this chapter.

Refactoring the Example Recall from Sect. 2 (Fig. 2) that the Important Messages Feature includes a single use case which, therefore, does not use inclusion and extension relations. Figure 3 represents the same system, although with a more structured use case model with inclusion and extension relations among use cases.

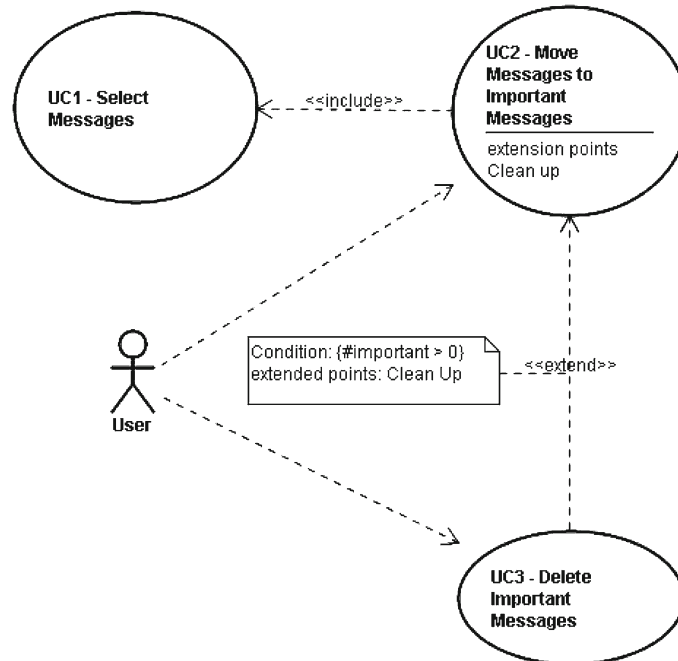

Fig. 3. Important messages—use case diagram

The selection of messages is a very common flow, potentially included by other use cases. We have created a use case (UC1-Select Messages) that describes the selection of messages (Fig. 3). The use case UC1 is defined as the two first steps of the main flow of the previous UC1 (Fig. 2a). The use case UC2 (Move Messages to the Important Messages folder) is defined as the last step of the main flow, and the first alternative step of the previous UC1 (Fig. 2a). The use case UC2 includes UC1. According to UML [Gro07], the use case that includes another is called the including use case (base behaviour), and the one that is referenced is called inclusion (or included use case). In our example, UC2 is the including use case and UC1 is the inclusion.

An extension point is a place mark inside the use case flow, labelled by a name, which allows extensions to add behaviour (optionally) to that point by referring to the label. As in our example, while moving messages it is also possible to (optionally) delete them (clean up); UC2 has an extension point labelled `Clean up`. The use case UC3 (Delete Important Messages) extends UC2 in the extension point `Clean up` and its flow is defined as the last step of the alternative flow of use case UC1 (Fig. 2b-bottom). As the extension point has an explicit state condition, it must be evaluated to decide whether the extension flow must be included or not in the extended use case. Let `important` be a state variable representing the current set of messages stored in the Important Messages folder. Thus, the behaviour of UC3 is added into UC2 if the cardinality of the set of important messages is greater than zero ($\#\texttt{important} > 0$). It is worth emphasising that this conditional behaviour extension can be properly captured only with a notion of state, as illustrated by this example. The use case that is extended by others is called the extended use case (base behaviour), and the one that extends it is called extension. For instance, in our example UC3 is the extension and UC2 is the extended use case.

An including use case can refer to many inclusions, and an inclusion can be referred to by several base use cases (many-to-many relationship). The same multiplicity applies to the extension relation. However, cyclic dependence between use cases is prohibited.

An actor is another important concept when describing use cases in UML. It represents an entity outside the system that interacts with the system. In our domain, it is often the user that interacts with the GUI of the implementation. Use cases that are associated with an actor are said to be active. An active use case is an independent functional unit that initiates its behaviour as soon as the first interaction of the actor is performed. As an example, in the use case diagram in Fig. 3, the use cases UC2 and UC3 are activated by the User (actor). Thus they are performed as independent functional units. We call auxiliary the use cases that are not active. An auxiliary use case is a dependent functional unit that performs the specified behaviour as a consequence of its relation (inclusion or extension) with some active use case. In our example, UC1 is an auxiliary use case that cannot be performed independently. It is performed only if UC2 is.

Data Definition Using the refactored example we present the new document template. We start by presenting how to define data in the document.

Figure 3 shows an expression that specifies the condition for the Use Case UC3 to extend UC2. So far, we have assumed the variable `important` was already declared and initialised. Now we show how to define data elements as types, constants, parameters and state variables. Data definitions can be shared among the use cases belonging to the same feature (feature scope), or be local to a use case (local scope).

In the use case document, types and expressions are written in a functional style similar to that available in CSP_M [Sca98]; this eases the reading and understanding of the translation into CSP_M. We explain the notation on demand. Nevertheless, in order to increase the practical appeal of the approach, we have developed a notation closer to natural language and a parser and translation into CSP_M [Bez11]; the integration of this notation into the ATG tool is a current work topic.

Often the types declared in the use case document are related to concepts in the scope of the requirements to be tested. Figure 4a exhibits data definitions valid for the scope of the Important Messages Feature. The types `Natural` and `Message` are declared using distinct constructors. The constructor `nametype` is used to associate a type expression with a name. In our example, the set comprehension representing the range of integers from 0 to 2 ({0..2}) is associated with the name (type) `Natural`. The constructor `datatype` is used to associate a type name with atomic values, or values prefixed by a tag followed by a dot. Thus, the values for the `Message` type are natural numbers prefaced with `M`. Constants are defined by associating a name with a value expression. As an example, the constant that represents the maximum number of messages stored in the Important Messages Folder (`MAX`) is assigned the constant value 2.

F1 – Important Messages

Data Definition

nametype Natural = {0..2}
datatype Message = M.Natural
MAX = 2

[FNAME]

var folder: ℙ (Message) = {M.0,M.1}
var selected: ℙ (Message) = {}
var important : ℙ (Message) = {M.2}

UC1 – Selecting Messages <<auxiliary>>

Main Flow

Step Id	Action	System State	System Response
1M	Go to folder. **\$FNAME\$**		All Messages are displayed. **\$FNAME\$**
2M	Select message(s). **[?x : ℙ(folder) – {{}}]**		Message(s) are highlighted. **[selected := x]**

(a) Important Messages data definition and use case UC1

UC2 – Moving to Important Messages Folder

Includes UC1@START
Extension points Clean up: 1A

Main Flow

Step Id	Action	System State	System Response
1M	Select "Move to Important Messages" option.	Message storage has enough space. **[#(important ∪ selected) ≤ MAX]**	"Message moved to Important Messages folder" is displayed. **[folder := folder – selected, important := important ∪ selected)]**

Alternative Flows

From Step: START
To Step: START

Step Id	Action	System State	System Response
1A	Select "Move to Important Messages" option.	Message storage has not enough space. **[#(important ∪ selected) > MAX]**	"Message storage has not enough space" is displayed. "Clean Up request" is displayed for X message(s). **[! #(important ∪ selected) – MAX]**

(b) Use case UC2

UC3 – Deleting Important Messages

Extends (**#important > 0,** UC2@Clean up)

Data Definitions

var selected: ℙ (Message) = **{}**

Main Flow

Step Id	Action	System State	System Response
1M	Select important message(s). **[?x : (important) – {{}}]**		Message(s) are highlighted. **[selected := x]**
2M	Clean up selected message(s).		Clean is performed. **[important := important – selected]**

(c) Use case UC3

Fig. 4. Important messages feature in the new template

Parameters and variables are very useful for the specification of features. Use case parameters are use case references to values defined outside the use case specification. A possible use of a parameter is to abstract data values that have a same treatment in the use case (for instance, moving a message to a message folder has the same behaviour regardless the name of the folder). Use case state variables are variables that are explicitly declared as part of the user interface and the state of the feature to be tested (for example, number of messages selected in the phone interface).

Parameters and state variables are declared using the syntax that follows. A parameter declaration has the form `[PName]`, where `PName` is the parameter name. The values a parameter can take are defined before the document is instantiated. The parameter can be of any type allowed to be referenced in the use case document. Each instantiation of a use case takes a unique value for each parameter to which it refers. In Fig. 4a the parameter `FNAME` defines the name of the source folders from where the messages can be moved to the Important Messages folder. The last declaration of a data element we present is that for state variables. The declaration of a state variable has the form `var VName:Type = Init`, where the tag `var` precedes the variable name (`VName`), its type

(Type) and initial value (Init). The expressions for type and initial value of a use case can be defined with primitives and constructors from CSP_M, in addition to the user defined types. The variables `folder`, `selected` and `important` are declared in Fig. 4a (top) with the same type, the powerset of values of the `Message` Type. These variables record, respectively, the messages kept in a source folder (`folder`), the messages selected to be moved (`selected`) and the messages stored in the Important Messages folder (`important`). The source folder is initialised with two messages, the selected message with no messages, and the Important Messages Folder with a single message.

Use Case Specification Provided all the necessary types, parameters and variables have been declared, we explain how to textually specify use cases possibly involving input, output and state variables. Figure 4 shows the use case document that specifies the use cases introduced in Fig. 3.

In the original version of the template, presented in the previous section, the fields `From Step` and `To Step` are used to simulate inclusion and extension relations. Instead, in the proposed template, relationships among use cases are specified with the fields `Includes`, `Extension points` and `Extends`. The possibility to relate the use cases through inclusion and extension relationships contributes to better structuring the document and making use case specifications more concise.

The field `Includes` indicates where the behaviour of the included use cases will be added in the including use case. As an example, in Fig. 4b UC1 is included after the START step of UC2. Consider $i_1, \ldots, i_k$ the list of inclusions. Each inclusion i_x ($1 \leq x \leq k$) has the form $uc_list@s$, where $uc_list = uc_1, \ldots, uc_w$ is a list of identifiers of the included use cases, and s is the step id after which the list of use cases will be added.

The field `Extension points` defines the extension points of a use case. For instance, in Fig. 4b UC2 defines an extension point labelled `Clean up` after the step 1A. Extensions associated with such an extension point will assume control after the step 1A and resume before its continuation (START step). Extension point declarations are separated by commas. Let $ep_1, \ldots, ep_k$ be the list of extension points of a use case. Each extension point ep_x ($1 \leq x \leq k$) has the form $e : s$, where e is the extension point label (unique in the list), and s is the step id after which the behaviour of an extension use case associated with e will be added.

The field `Extends` specifies the extension points where an extension use case adds behaviour. As an example, Fig. 4c shows that UC3 extends UC2 in the extension point `Clean up` provided the Important Messages Folder is not empty (#important > 0). Consider $ext_1, \ldots, ext_w$ the list of extensions originated from a use case. Each extension ext_x ($1 \leq x \leq w$) is a tuple of the form $(c_x, uc_x@e_x)$, where c_x represents the (optional) condition for the extension, and $uc_x@e_x$ the extension point e_x in the extended use case uc_x where the behaviour is added.

As it can be seen in the running example (Fig. 4), use cases can only be related through inclusion and extension relations. As a consequence, the `From Step` and `To Step` fields become exclusive for referring to flow steps that are local to the use case. Furthermore, in the document template, an auxiliary use case has its title tagged with the suffix <<`auxiliary`>>, as exemplified by use case UC1 (Fig. 4a).

A state based use case can later be instantiated individually for each value of its parameters. The form for introducing a parameter `PAR` is by referring to its name enclosed within the token `$` (that is, `$PAR$`). For instance, the specification of UC1 is parametrised by `FNAME` (Fig. 4a). One can observe that the steps of the use case UC1 in Fig. 4a differ in steps 1M and 2M from the corresponding steps in Fig. 2a. Particularly, the references to the Inbox Folder, in Fig. 2a, were replaced by a reference to the `FNAME` parameter. Such a generalisation is possible since the specification for selecting messages is the same, regardless of the folder in which it is performed (for instance, Inbox or Outbox).

Another important facility provided by the new template are constructions to describe input, output, guard and state update, as a complement to the textual description. These constructions are essential to describe more elaborate flows, as those controlled by the dynamic changes in the system state. For example, UC2 (in Fig. 4b) states that the alternative flow can be repeated indefinitely, as far as the number of messages to be moved exceeds the limits. This could not be precisely specified without an explicit notion of state. For keeping compatibility with the original template, in the new template data fields are enclosed between square brackets and are not part of the CNL sentences. Hence, the proposed template extensions have no impact on the CNL standard [Tor06]. The new data fields are the following.

Inputs are supplied by actors and associated with the step action. For instance, the input `x` in step 2M of UC1 represents the set of (non-empty) messages selected by the user in the source folder. Formally, this input takes a value from the set $\mathbb{P}(\texttt{folder}) - \{\emptyset\}$. The general form for the inputs is the list $?i_1 : S_1, \ldots, ?i_k : S_k$, where i_x is the input name (unique in the step) and S_x the expression that defines the set of values i_x can take, for $1 \leq x \leq k$. Inputs are used in the same sequence they are defined, so an input i_z can reference values from i_w, provided $z > w$.

State guard is a condition on the values of variables; it is defined as a boolean expression placed in the system state column. For instance, in the UC2 step 1M the guard `#(important ∪ selected) ≤ MAX` specifies the condition to move the select messages to the Important Messages folder: the cardinality of the union of the Important messages and the selected messages must not exceed the maximum capacity of the Important Messages Folder. If the condition is false, the step is not performed and the use case does not progress from the step.

Variable assignments specify an update in the system state after the finalisation of a step. For instance, in the step 2M of the UC1 the input `x` is assigned to the set of selected messages, whereas the other variables (`folder` and `important`) remain unchanged. A sequence of assignments is allowed, as, for instance, the assignments in the step 1M of UC2 that update the values of the variables `folder` and `important`. The general form for the sequence of assignments is $v_1 := e_1, \ldots, v_k := e_k$, where v_x is a variable and e_x an expression whose value is assigned to v_x, for $1 \leq x \leq k$. Variables that are not in the sequence are assumed unchanged.

Output values are associated with the system response. For instance, `!#(important ∪ selected) - MAX` is the output expression in step 1A of the use case UC2 that completes the system response message by calculating the number of messages that exceed the maximum capacity of the Important Messages Folder. The general form is the list of output expressions $!e_1, \ldots, !e_k$.

Once all elements of the new template have been introduced, we can explain the complete specification for the Important Messages feature in Fig. 4. The auxiliary use case UC1 specifies the selection of messages from a source folder whose name is parametrised (`FNAME`). The second step of this use case updates the set of selected messages with the messages that are input. The use case UC2 includes UC1 after the START step, and defines an extension point named `Clean up`. The main flow of UC2 specifies the successful attempt of moving the selected messages to the Important Messages Folder. After moving the messages, the selected messages are removed from the source folder and added into the target. On the other hand, the alternative flow of UC2 specifies the unsuccessful attempt to move messages. The guard of step 1A is a complement for that of step 1M, so they are mutually exclusive. If the guard of step 1A holds, then a message is displayed indicating that a cleanup is required so that the selected messages can be moved; the least number of messages to be deleted is indicated. The use case UC3 extends UC2 in the `Clean up` extension, if there is some message in the Important Messages Folder. This use case defines a local variable named `selected` that is initialised with an empty set. If a local variable has the same name as a global (overloading) variable, references to the variable name will refer to the local one. In UC3 the messages selected from the Important Messages Folder are assigned to a local variable (step 1M), and the same selected messages are deleted from that folder.

4.2. Overview of the CSP model

This section presents the CSP model for the proposed state-based use case template. Figure 5 displays the structure of the CSP model for the Important Messages Feature presented in Fig. 4. The process *System* models the behaviour of a set of features of an application, and reflects the document structure. Basically, for each document feature a CSP process is generated. In our example, the single feature F1 gives rise to a process with the same name. Accordingly, each use case of a feature is modelled by a process. As originally conceived, CSP processes do not have an explicit notion of variable or state. A simple alternative to model state information is through process parameters. Another alternative, adopted in this work, is to simulate a memory as a separate process. Such a process keeps the state of variables and enables others to read and modify the values by synchronising in common events [Ros98, RH07]. Likewise, if the feature contains variables, additionally to the use case processes, another process is created to specify the feature state; this process plays the role of a memory that is global to all feature use cases. A memory process keeps variable values and allows control flow processes to read from and write to the memory. In our example, F1 is formed of three use case processes (UC1, UC2 and UC3) and one additional process to model the feature memory (F1_MEM). If a use case does introduce any variable its model reduces to the process that specifies its flow (for instance, UC1 is just UC1_FLOW). Otherwise, the model is the parallel composition of the flow and the memory processes (for instance, UC3 is defined by the composition of UC3_FLOW and UC3_MEM). Flow and memory processes communicate through special events that enable reading (get) and updating the memory (set). Such events are shared among flow and memory processes. A use case memory is only accessed by the use case flow, while a feature memory is shared among the feature use cases.

This modular structure has allowed us to devise an incremental strategy to translate the templates into CSP processes. As another advantage we could easily extend the test selection approach, originally designed to control flow processes, to consider particular states of the model.

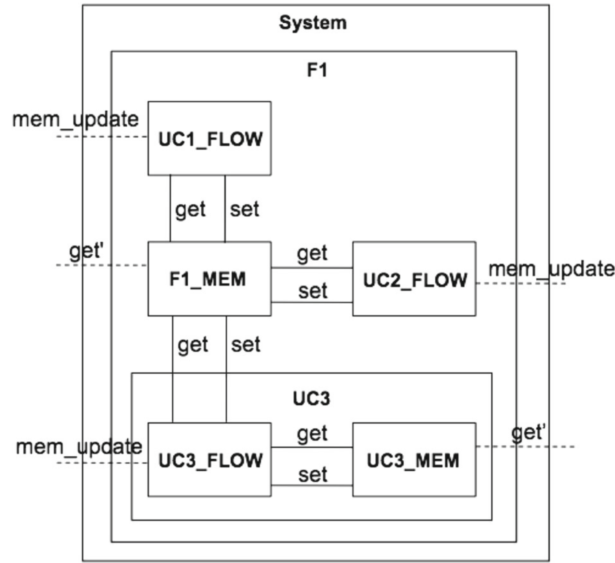

Fig. 5. Important messages model with get' events

01	$nametype\ \ tF1_Natural = \{0..2\}$
02	$datatype\ \ tF1_Message = M.tF1_Natural$
03	$F1_MAX = 2$
04	
05	$nametype\ \ tF1_FNAME = \{\ldots\}$
06	
07	$datatype\ \ Var = F1_folder \mid F1_selected \mid F1_important \mid F1_UC3_selected$
08	
09	$datatype\ \ Type = F1_Messages.Set(tF1_Message)$
10	
11	$bF1_MEMinit = \{(F1_folder, F1_Messages.\{M.0, M.1\}),$
12	$(F1_selected, F1_Messages.\{\}),$
13	$(F1_important, F1_Messages.\{M.2\})\ \}$

Fig. 6. CSP model for types, constants and variables

In the sequel we incrementally present the translation from templates to CSP processes. This translation is illustrated by our running example. The CSP notation used includes a few elements from CSP_M: constructors for declaring types and channels. Apart from these elements the models are represented using the CSP notation, which provides an easier reading.

4.3. Types, constants and variables

We start with the translation of types, constants and variables declared in the template into CSP. The general rule for the names in the CSP model is that elements valid in the scope of a feature f are prefixed with $f_$, and those valid in the scope of a use case uc of the feature f are prefixed with $f_uc_$. In this way we avoid eventual name clashing in the CSP model.

Data elements declared in Fig. 4a are translated into the specification depicted in Fig. 6. The CSP model for types and constants (lines 1–3) is a straightforward translation of the data declarations: the names are prefixed to indicate the scope of the elements. Moreover, type identifiers are preceded with the character t. Each parameter is translated similarly to a type, but the specific values (instantiation) are left open for the user to provide. The parameter for our example is translated into the nametype $nametype\ \ tF1_FNAME = \{\ldots\}$ (line 5) which must be completed before the document instantiation.

01	$channel\ \ get, set : Var.Type$
02	
03	$Mcell(v, val) = get!v!val \rightarrow Mcell(v, val)$
04	$\quad\quad\quad\quad\quad\quad \Box$
05	$\quad\quad\quad\quad\quad\quad set!v?val' : range(tag(val)) \rightarrow Mcell(v, val')$
06	
07	$Memory(binding) = \vert\vert\vert (v, val) : binding \bullet Mcell(v, val)$

Fig. 7. CSP model for memory

Lines 7–9 of Fig. 6 depict the CSP model for the variables, for a type that is the union of all introduced types, and for the memory of the Important Messages Feature. Each variable declared in the document (in the scope of a feature or local to a use case) is enumerated in the datatype Var. In the specification above the datatype Var enumerates four variables: the first three in the scope of the feature (F1_folder, F1_selected and F1_important), and the last one (F1_UC3_selected) in the scope of the use case UC3 (see Fig. 4c). Furthermore, each variable type is enumerated in the datatype $Type$, whose purpose is to represent the union of all the relevant types of state variables; this is then used as the type of the variables placed in the memory. The structure of $Type$ eases the representation of a very simple abstract memory in CSP, which treats uniformly variables of different types, as can be seen in the next section. Each type is declared by a tag that identifies the type and the separator '.' followed by the set of values for the type. The datatype $Type$ produces values in the form $typeTag.typeValue$, which are denominated tagged values. In the specification fragment above, $Type$ enumerates the (single) type of the variables declared in the Important Messages feature. This variable is tagged with the identifier $F1_Messages$ and values in the superset of message sets. The value $F1_Messages.\{\}$ is a trivial example of a tagged value from $Type$, and represents the empty set of messages. From now on, whenever we refer to the value of a variable we implicitly refer to its tagged value.

The initial memory environment of a feature f, say $bf_MEMinit$, maps the variables in the scope of f with their respective initial (tagged) values. The mapping of the variables to their values is specified as the set of tuples of the form (variable, tagged valued). In lines 11–13 of Fig. 6, $bF1_MEMinit$ is the initial binding for the three variables in the scope of the Important Messages Feature. The binding of variables local to a use case (for instance, the binding of the variable $F1_UC1_selected$ from UC3) is specified analogously and exemplified later.

4.4. Memory model

Figure 7 shows the CSP specification for the memory model. The CSP channels get and set (line 1) are used to allow read and write access to the memory. These channels are able to communicate the pairs (variable, tagged value) whose values come from the already defined datatypes Var and $Type$, respectively. Based on these channels we can define the process that models a memory.

There are several alternatives for representing state information in CSP, ranging from a centralised map (which associates variables with their respective values) to an interleaving of processes, each one concerned with the store of a single variable. As discussed in [Ros11], the latter has the advantage of a possible concurrent access, and turns out to be a very efficient solution regarding FDR analyses. Particularly, the interleaving model helps the compression algorithms built in FDR to minimise the state space while analysing interleaved CSP models. This is an important issue to consider, since parallel composition leads to exponential growth in the space state. Therefore we adopted this approach for the purpose of an efficient analysis. Nevertheless, currently, we model only sequential applications. Concurrent applications (for example, several use case processes running in parallel) are discussed as a topic for future work. Consider the function $tag(tag.val)$ that yields the tag of a value $tag.val$, and, the function $range(type)$ that yields the set of values of type $type$. The memory process for a single variable (memory cell) is depicted in lines 3–5 of Fig. 7. The non-terminating recursive process $Mcell(.)$ carries the variable v and its current value val as process parameters.

It behaves as the choice between the process

$get!v!val \rightarrow Mcell(v, val)$

(whose prefix offers the current value of v and recurses) and the process

$set!v?val' \rightarrow Mcell(v, val')$

that accepts a new value val' for v and recurses with this new value.

The memory process is the interleaving among memory cells (see the replicated construction of CSP in Sect. 3.1), each one representing a variable with its corresponding value from an initial *binding* (line 7 of Fig. 7). The read/write access to a particular variable is performed independently of the others, as already explained. This parametrised process can be instantiated considering the particular state space of a use case or feature model. For instance, the process $F1_MEMORY = Memory(bF1_MEMinit)$ is the memory model of Important Messages Feature, which enables the reading and the writing of the variables $folder$, $selected$ and $important$.

4.5. Flow and memory composition

The design of the processes that specify the use case flows is directly influenced by the way they are composed with memory processes. Thus, before showing the details of the CSP model for the use case flows we show how they are composed with the memory.

A use case process, say uc, has a subprocess for its control part, say uc_FLOW, possibly parametrised by a list of parameters, say uc_params, as well as by the list of extension points defined in the use case, say $uc_extensions$. The behaviour of the use case UC is modelled in CSP as the parallel composition of its control part (uc_FLOW) and its memory, say uc_MEMORY, with synchronisation on the events from memory, say α_{uc_MEMORY}. The alphabet of the memory is internal to control and memory processes, so they are hidden. Consequently, get and set events are synchronised between the flow and the memory processes and are invisible to the environment. When a use case process involves a parametrised flow process, it must also be parametrised.

Because the memory process (uc_MEMORY) is a recursive, non-terminating process, its direct composition with a flow process would also lead to non-termination; a parallel composition successfully terminates only when all its argument processes do terminate. Since we want to preserve the termination of flows (represented in CSP with the *Skip* process), we need to force the composition of flow and memory to terminate whenever the flow does. Consider the special event $success$ that is not in the alphabet of the memory, neither in the use case alphabet. Such an event is used to define the process $END = success \rightarrow Skip$, which communicates $success$ and then terminates successfully (*Skip*). The following CSP process models the behaviour of the use case UC. It terminates whenever the flow does.

$$
\begin{array}{l}
uc(parameters, extensions) = \\
\quad (uc_FLOW(uc_params, uc_extensions);\; END \\
\qquad [|\alpha_{uc_MEMORY} \cup \{success\}|] \\
\quad (uc_MEMORY \bigtriangleup END)) \setminus (\alpha_{uc_MEMORY} \cup \{success\})
\end{array}
$$

In the process above whenever the flow terminates with success (on the left-hand side of the parallel composition) it behaves as the END process, which is $success \rightarrow Skip$. Because $success$ is in the synchronisation alphabet of the parallel composition, the event $success$ can only be communicated if the same event is offered on the right-hand side of the parallel composition, and vice-versa. At this point, the only possible behaviour for $uc_MEMORY \bigtriangleup END$ is $success \rightarrow Skip$. As a consequence, the resulting behaviour of the composition becomes $(success \rightarrow Skip) \setminus \{success\}$; since the only visible event is hidden ($success$) it is equivalent to *Skip* meaning the the entire parallel composition terminates.

4.6. Use case model

Recall from Sect. 3.1 that the use case control flow is modelled as a CSP process, composed of locally defined processes, one for each step. Now we augment this representation to consider the new elements added to the template. Let uc_START be the process that models the start step, and uc_s the one that models a step s. The following description represents the constituents of the process that models the flow of the

use case UC.

$$
\begin{array}{l}
uc_FLOW(uc_params, uc_extensions) = let \\
\quad uc_START = inclusions(start_params); \\
\qquad\qquad\qquad continuations_st \\
\quad uc_s = inclusions(s_i_params); \\
\qquad readVars \rightarrow condition \; \& \; inputs \rightarrow \\
\qquad action(s_a_params) \rightarrow \\
\qquad response(s_r_params) \rightarrow \\
\qquad outputs \rightarrow updateVars \rightarrow Skip; \\
\qquad (s_extensions \; \Box \; Skip); \; continuations_s \\
\qquad \ldots \\
\quad within \;\; uc_START
\end{array}
$$

The process that models the start step behaves as the inclusions in the start step, represented by the process $inclusions$, which is sequentially composed with the process that models the continuations of this step, given by the process $continuations_st$. Likewise, the CSP process for any other step (uc_s) behaves initially as the process $inclusions$ and finally as $continuations_s$. The default behaviour for the $continuation_st$ process is the first step of the main flow. If there are steps coming from the start step, the behaviour is modelled as the choice of the first step with those that follow after the start step. Similarly, the default behaviour of $continuations_s$ is the subsequent step in the flow (if it exists). If there are steps coming from the step s, the behaviour is modelled as the choice of the subsequent step with those that follow after the step s. In between $inclusions$ and $continuations_s$ the behaviour is sequentially modelled by the following: $readVars$ reads from memory the referenced variables, $condition$ models the step condition, $inputs$ models the step inputs, $action$ models the step action, $response$ models the step response, $outputs$ models the step outputs, $updateVars$ models variable assignments, and $s_extensions$ models the extension points in step s. The constituents named $readVars$, $inputs$, $outputs$ and $updateVars$ are meta-elements representing sequences of prefix events. The extensions for the use case steps are grouped into parameters in the uc_FLOW process ($uc_extensions$). By default, the first step of the use case main flow comes from the start step.

According to the UML 2.0 specification [Gro07], the including use case can see the inclusions and may depend on their effects. Hence, in our CSP model the behaviour of the included use cases comes after the step indicated in the feature document and before the subsequent step. On the other hand, the extended use case cannot see the extensions, so the extensions are parametrised. As system conditions now use state variables, their values must be read in advance from the process memory. Furthermore, the system condition enables or blocks the step action, consequently the step condition comes before the step action.

Included use cases ($inclusions_s$) and action/response ($action$ and $response$) can be defined in terms of parameter values, and are parametrised. The lists $start_params$, s_i_params, s_a_params and s_r_params specify, respectively, the parameters that are referenced in the inclusions of the start step, in the inclusion of the step s, and in the action and response of the step s. The parameter lists for the use case steps are grouped into a single parameter, uc_params, in the uc_FLOW process.

Now we explain each element of a use case flow incrementally based on the use cases in Fig. 4.

Action and Response Similarly to Sect. 3.1, the action and the system response of a step are modelled as events. We further explain how actions and responses are modelled by using our running example. The complete CSP model for the use case UC1 (see Fig. 4a) is shown in Fig. 8.

A step action or response is represented by a CSP event that can communicate (input or output) values specified in the respective step. In CSP, events associated with values are obtained by instantiating channel communication. Therefore, channel declarations must be introduced before we can model actions and responses in terms of events. For instance, if we need to represent a step action sa as an event $sa.v_1.v_2. \; \ldots \; .v_k$ we need to introduce the channel declaration $channel\ sa : t_i. \; \ldots \; .t_k$, where t_x is the set of values that sa can communicate in the xth position, for $1 \leq x \leq k$. For instance, the channels for the use case UC1 in Fig. 4a are depicted in lines 1–4 of Fig. 8. The channels $goTo$, $msgsDisp$, $selectMsgs$ and $msgsHighlighted$ specify actions and responses of the steps 1M (two first channels) and 2M (two last channels), which communicate the values of the parameter $F1_FNAME$ (folder name).

```
01 channel goTo : tF1_FNAME
02 channel msgsDisp : tF1_FNAME
03 channel selectMsgs : tF1_FNAME
04 channel msgsHighlighted : tF1_FNAME
05
06 channel  in_F1_UC1_1A_x : range(F1_Messages)
07
08 F1_UC1_FLOW(FNAME) = let
09
10    F1_UC1_START = F1_UC1_1M
11
12    F1_UC1_1M = goTo.FNAME → msgsDisp.FNAME →
13       Skip; (F1_UC1_2M)
14
15    F1_UC1_2M = get!F1_folder?folder : range(F1_Message) →
16       in_F1_UC1_1A_x?x :
17         {F1_Messages.v | v ∈ (ℙ(value(folder)) − {∅})} →
18       selectMsgs.FNAME → msgsHighlighted.FNAME →
19       set!F1_selected!x → mem_update →
20       Skip
21
22 within  F1_UC1_START
23
24 F1_UC1(FNAME) = F1_UC1_FLOW(FNAME)
```

Fig. 8. CSP model for UC1

Consider the rule to define continuations is the same presented in Sect. 3.1. Hence, the flow process of the use case UC1 (lines 8–22 of Fig. 8) is parametrised by `FNAME` and has three subprocesses: the first models the start step ($F1_UC1_START$) and continues as the process of step 1M; the second models the step 1M ($F1_UC1_1M$), which is also parametrised by `FNAME` and continues as the process of the step 2M; the third is the model of the step 2M, which is detailed in the sequel. Since UC1 does not have local variables (thus no memory), the process $F1_UC1(FNAME)$ is equivalent to its flow process (line 24 of Fig. 8).

Variable Reading As already mentioned, variable values are kept by memory processes. Consequently, before they can be used they must be put in scope. For instance, the step 2M of the use case UC1 refers to the value of the variable *folder* in the input field, thus this variable must be put in the context of the step 2M, as shown in line 15 of Fig. 8.

As already seen, a use case process is composed in parallel with its memory process (if it has local variables). In addition, the use case processes are in parallel with the process that represents the feature memory (with synchronisation set formed of *get* and *set* events). In our example, the process $F1_UC1$ does not have a local state, but it is composed in parallel with the feature memory process $F1_MEM$ to form the feature model of F1 (presented later). The prefix $get!F1_folder?folder : range(F1_Message) \rightarrow$ communicates values of the type $F1_Messages$. Since this communication synchronises with the process that represents the feature memory, which offers an event $get!F1_folder!value$, such that $value$ is the current value for the variable $F1_folder$, the value bound to *folder* is the current value of the variable $F1_folder$. As a result of this synchronisation, the free occurrences of the name *folder* in the process $F1_UC1_2M$ are replaced by the current value of $F1_folder$.

In general, the reading of a list of distinct variables, say $var_1, \ldots, var_k$, is modelled as

$$get!var_1?var_1 \rightarrow \cdots \rightarrow get!var_k?var_k$$

The order in which the variables are read does not matter because they are used only after all of them are in context. Therefore, the reading can alternatively be modelled in parallel, but the sequential model is suitable for our application and turns out to be more efficient for analysis. Of course, this is valid because we are dealing only with sequential applications, as already explained. The extension for dealing with concurrency is discussed as a topic for future work.

Continuing the translation of the step 2M we have the CSP model for the input value `x`; see Fig. 4a.

Input The values for an input variable are directly specified in CSP by a channel, which is used to specify the environment option for a particular value inside the range of the input. As an example, the values for the input `x` in step 2M of UC1 are defined by the channel $in_F1_UC1_1A_x$ specified in line 6 of Fig. 8.

According to the channel $in_F1_UC1_1A_x$, the values of the input `x` are those of $F1_Messages$. Each expression in the use case document translated to CSP deals with the value part of tagged values. Let $value(Type.v)$ be the function that retrieves the value component (v) of a tagged value. Such a channel is used to model the input of the process $F1_UC1_2M$. The lines 16–17 of Fig. 8 specify the input `x`, the action of selecting messages and the respective response (messages are highlighted). The value of `x` is a non-empty subset of messages from the value of `folder`; it is read from the environment and put in the context of the process $F1_UC1_2M$. This enables subsequent elements to use the value associated with `x`. The action and response are parametrised by `FNAME`.

The list of inputs has the form $?i_1 : S_1, \ldots, ?i_k : S_k$. Consider $type(exp)$ the function that yields the type (tag) of the expression exp, and, that the variables referred to by the expression are already in the context of the step process. Also consider the function $val(exp)$ that yields the current value of exp. Thus, the CSP model for the list of inputs in a step s is

$$
\begin{aligned}
&in_s_i_1?i_1 : \{type(S_1).v \mid v \in val(S_1)\} \rightarrow \\
&\cdots \rightarrow \\
&in_s_i_k?i_k : \{type(S_k).v \mid v \in val(S_k)\}
\end{aligned}
$$

where $in_s_i_x.val$ is an event originated from the channel

$$channel\ in_s_i_x : range(type(S_x))$$

for $1 \leq x \leq k$.

Variable Update Due to the representation of variables as memory processes, the CSP model for assigning a value to a variable is very similar to the one for reading. However, while in the case of reading the use case process gets the value from memory, an assignment, represented by the event set, indicates that the communicated value must update the value of the communicated variable identifier. As an example, consider the last two lines of the step process $F1_UC1_2M$ (lines 19–20 of Fig. 8). The communication that prefixes the successful termination updates the variable `x` with the set of selected messages ($F1_selected$). The auxiliary event mem_update indicates the finalisation of a sequence of assignments; it does not belong to the input neither to the output events of the CSP model. Such an event is used in the selection of test scenarios based on memory states, as discussed in the next section.

In general, the CSP model for a sequence of assignments $v_1 := e_1, \ldots, v_k := e_k$ is

$$
\begin{aligned}
&set!v_1!type(v_1).val(e_1) \rightarrow \\
&\ldots \rightarrow \\
&set!v_k!type(v_k).val(e_k) \rightarrow \\
&mem_update \rightarrow Skip
\end{aligned}
$$

Similarly to variable reading the order in which the variables are updated does not matter, as we are dealing only with sequential processes.

Inclusion and Step Condition Let $i_1, \ldots, i_k$ be the list of use cases included in some step uc_s. The CSP model for the inclusions is the sequential composition $i_1; \ \ldots; \ i_k$. As an example, consider the partial translation of the use case UC2 presented in Fig. 9. The process flow of the use case UC2 ($F1_UC2_FLOW$) is parametrised by `FNAME` as well as by the use case extension ($F1_UC2_CleanUp$). Its start step includes the use case UC1 that is sequentially composed with the continuations of the step, the choice: $F1_UC2_1M \,\Box\, F1_UC2_1A$.

An advantage of representing state explicitly is that we can model conditions as expressions on state variables ($condition$), which provides a more accurate representation than abstracting state conditions as CSP events, which is the case with the restricted model presented in Sect. 3. Furthermore, it also allows us to perform test selection based on particular states of interest. Hence the sentences that represent the system state in the use case template are used here exclusively for documentation and are not part of the CSP model.

```
01 channel out1_F1_UC2_1A : tNaturals
02
03 F1_UC2_FLOW(FNAME, F1_UC2_CleanUp) = let
04
05    F1_UC2_START = F1_UC1(FNAME);
06       (F1_UC2_1M □ F1_UC2_1A)
07
08    F1_UC2_1M =
09       get!F1_important?important : range(F1_Messages) →
10       get!F1_selected?selected : range(F1_Messages) →
11       get!F1_folder?folder : range(F1_Messages) →
12       #(value(important) ∪ value(selected)) ≤ F1_MAX &
13       selMoveToIMOpt → msgMovedToIMDisp →
14       set!F1_important!F1_Messages.
15         (value(important) ∪ value(selected)) →
16       set!F1_folder!F1_Messages.
17         (value(folder) − value(selected)) →
18       mem_update →
19       Skip
20
```

Fig. 9. CSP model for UC2—first part

```
20
21    F1_UC2_1A =
22       get!F1_important?important : range(F1_Messages) →
23       get!F1_selected?selected : range(F1_Messages) →
24       #(value(important) ∪ value(selected)) > F1_MAX &
25       selMoveToIMOpt → cleanUpReqDisp →
26       out1_F1_UC2_1A!
27         (#(value(important) ∪ value(selected)) − F1_MAX) →
28       Skip; (F1_UC2_CleanUp □ Skip);
29       F1_UC2_START
30
31 within F1_UC2_START
32
33 F1_UC2(FNAME, F1_UC2_CleanUp) =
34    F1_UC2_FLOW(FNAME, F1_UC2_CleanUp)
```

Fig. 10. CSP model for UC2—last part

Let $guard$ be the state guard specified in the system condition of a step s. The model for the system condition of s is $val(guard)$ and the guarded process $val(guard)$ & $action \rightarrow \ldots$ behaves as the subsequent processes $action \rightarrow \ldots$ if the required state is true, otherwise, it deadlocks ($Stop$). For instance, the CSP model for the step 1M of UC2 in Fig. 9 initially places the referred variables (`important`, `selected` and `folder`) in the context using get events, and the subsequent behaviour is guarded by the state guard $\#(value(important) \cup value(selected)) \leq F1_MAX$. If the guard holds, the subsequent action and response are performed and the system state is updated accordingly.

Output The specification of $outputs$ is very similar to that of $inputs$. The values for an output are directly specified in CSP by a channel, which is used to communicate the value of an output expression.

Consider the remaining part of the CSP model for UC2 depicted in Fig. 10. As an example of output, the values for the output of the step 1A of UC2 are defined by the channel $out1_F1_UC2_1A$ (line 1 of Fig. 9). In the step 1A (lines 21–29 of Fig. 10), firstly, the referenced variables are read from memory; the subsequent behaviour is guarded by the expression $\#(value(important) \cup value(selected)) > F1_MAX$. In sequence, after communication of the step action and response, the output expression $\#(value(important) \cup value(selected)) - F1_MAX$ is communicated, indicating the number of messages that need to be deleted in order to allow moving the selected messages.

The CSP model for the list of outputs $!e_1, \ldots, !e_k$ of the use case step uc_s is

$out1_uc_s!val(e_1) \rightarrow$
$\ldots$
$outk_uc_s!val(e_k) \rightarrow$

where for each output out_x there is a channel

$channel\ outx_uc_s : type(e_x)$

for $1 \leq x \leq k$.

Extension Point Let $\{ep_1, \ldots, ep_k\}$ be the set of extension points defined in the step uc_s. Since the extensions added to a same extension point are performed individually, the CSP model for the extension points is given by

$(\Box\, e : \{ep_1, \ldots, ep_k\} \bullet e) \Box\, Skip$

If there is no extension associated with the extension points, the behaviour of ep_x $(1 \leq x \leq k)$ is equivalent to $Stop$, and the behaviour of the above process reduces to $Skip$. The CSP model for the extensions, particularly the choice with $Skip$, shows that extensions are additional branches in the step behaviour, so the step behaviour can always continue without performing extensions. Let ext be an extension point of the use case uc, in the feature f. The parameter f_uc_ext is used for referencing ext in the control flow of the process f_uc_FLOW. As an example, observe the fragment of the CSP model for UC2 in lines 28–34 of Fig. 10. In this fragment of the step 1A, the parameter $F1_UC2_CleanUp$ represents the behaviour of a use case that extends the use case UC2 in the `Clean up` extension point. Subsequently, the step 1A continues as the start step ($F1_UC2_START$). Finally, since the use case $F1_UC2$ has no local state, it is equivalent to its flow: $F1_UC2_FLOW(FNAME, F1_UC2_CleanUp)$.

Extension We have presented the CSP model for extension points, now we show how the extensions and their associations with extension points are modelled. First we show the CSP model for the use case UC3 depicted in Fig. 4c, which is our example of use case extension. This model is presented in Fig. 11.

The channel $in_F1_UC3_1M_x$ defines the values for the input `x` in the step 1M of UC3. The start process $F1_UC3_START$ is simply modelled by its continuation ($F1_UC3_1M$). The process for the step 1M ($F1_UC3_1M$) reads the set of important messages from memory (variable `important`), inputs a value in `x` (a non-empty subset of messages from important messages), communicates the step action and response, updates the set of selected messages (variable `selected`) and continues as step 2M. The process for the step 2M ($F1_UC3_2M$) reads the variables `important` and `selected`, communicates the step action and response, updates the variable important and finishes.

The process $F1_UC3$ has, in addition to the flow, a memory process that models the local variable $selected$ (see lines 25–32 of Fig. 12).

The model for the extension is first defined, and later connected to the extended use case. For instance, the model for the extension the use case UC3 performs in the extension point `Clean up` of the use case UC2 is presented in lines 34–36 of Fig. 12. The process $F1_UC3_Ext_F1_UC2_Cleanup$ reads the variables referenced in its condition ($F1_important$) and behaves as F1_UC3 if the extension condition holds (the Important Messages Folder is not empty). This process represents the behaviour of the extension itself, which is not yet connected into the extension point `Clean up` in the use case UC2. In the use case diagram in Fig. 3, this process represents the circle of UC3 and the outgoing arrow, which is associated with a note containing the extension point to be connected and the required condition. In the CSP model, the connection of such an arrow with the use case UC2 is modelled as the process $F1_UC3_Ext_F1_UC2_Cleanup$ as an instance of the extension parameters ($uc_extensions$) in the process $F1_UC2$, that is $F1_UC2(FNAME, F1_UC3_Ext_UC2_Cleanup)$. This replacement specifies that the behaviour of the extension point `Clean up` in UC2 can be that of UC3, if the condition extension condition holds (`#important > 0`). This replacement (connection) is performed in every occurrence of the process $F1_UC2$ in the CSP model.

```
01 | channel in_F1_UC3_1M_x : range(F1_Messages)
02 |
03 | F1_UC3_FLOW = let
04 |
05 |    F1_UC3_START = Skip; F1_UC3_1M
06 |
07 |    F1_UC3_1M =
08 |       get!F1_important?important : range(F1_Messages) →
09 |       in_F1_UC3_1M_x?x :
10 |          {F1_Messages.v | v ∈ ℙ(value(important)) − {∅}} →
11 |       selectImpMsgs → msgsHighlighted_ →
12 |       set!F1_UC3_selected!F1_Messages.value(x) →
13 |       mem_update → Skip; F1_UC3_2M
14 |
15 |    F1_UC3_2M =
16 |       get!F1_important?important : range(F1_Messages) →
17 |       get!F1_UC3_selected?selected : range(F1_Messages) →
18 |       cleanUpMsgs → cleanUpPerformed →
19 |       set!F1_important!F1_Messages.
20 |          (value(important) − value(selected)) →
21 |       mem_update → Skip
22 |
23 | within F1_UC3_START
24 |
```

Fig. 11. CSP model for UC3—first part

```
24 |
25 | bF1_UC3_MEM = {(F1_UC3_selected, F1_Messages.{})}
26 |
27 | F1_UC3_MEMORY = Memory(bF1_UC3_MEM)
28 |
29 | F1_UC3 = (F1_UC3_FLOW; END
30 |                 |[ α_F1_UC3_MEM ∪ {success} ]|
31 |               (F1_UC3_MEMORY △ END))
32 |                  \ (α_F1_UC3_MEM ∪ {success})
33 |
34 | F1_UC3_Ext_F1_UC2_Cleanup =
35 |    get!F1_important?important : range(F1_Messages) →
36 |    #(value(important)) > 0 & F1_UC3
```

Fig. 12. CSP model for UC3—last part

The CSP model for an arbitrary list of extensions is as follows. Consider $ext_1, \ldots, ext_w$ the list of extensions the use case uc performs. The CSP model of an extension $ext_x = (c_x, uc_x@e_x)$ performed by uc is

$$
\begin{aligned}
&uc_Ext_uc_x_e_x(uc_params, uc_extensions) = \\
&\quad readVars_x \rightarrow condition_x \ \& \\
&\qquad uc(uc_params, uc_extensions)
\end{aligned}
$$

for $1 \leq x \leq w$. The meta-element $readVars_x$ models the reading of the variables referenced in the condition c_x, and the expression $condition_x$ models c_x. They are modelled in the same way as $readVars$ and $condition$ of the use case flow, which were presented above.

Consider the function $extensions(uc, ep) = \{uc'_Ext_uc_ep\}$, which yields the set of CSP processes that model the extensions of the use case uc in the extension point ep, and the process $connection(uc, ep)$, defined as $\Box\, e : extensions(uc, ep) \bullet e$, which behaves as the choice among the extensions of uc in the extension point ep. Moreover, consider that $ep_1, \ldots, ep_k$ is the list of extension points of the use case uc. The signature of the use case process after connecting extensions into the respective extension points is

$$uc(uc_params, connection(ep_1), \ldots, connection(ep_k))$$

Use Case I/O process The complete model for a use case is given by an I/O process. Let uc be the CSP process that models a use case uc, $A_{I_{UC}}$ its input alphabet and $A_{O_{UC}}$ its output alphabet. Additionally, $actions(UC)$ represents the events that models the actions of the use case UC, $inputs(UC)$ the inputs, $responses(UC)$ the responses and $outputs(UC)$ the outputs. The tuple $M_{UC} = (UC, A_{I_{UC}}, A_{O_{UC}})$ is the I/O process for UC, such that $A_{I_{UC}} = actions(UC) \cup inputs(UC)$ and $A_{O_{UC}} = responses(UC) \cup outputs(UC)$. For instance, the I/O process of the use case UC2 is

$$\begin{aligned}
A_{I_{UC2}} &= \{selMoveToIMOpt\}\\
A_{O_{UC2}} &= \{msgMovedToIMDisp, cleanUpReqDisp\} \cup \{|\ out1_F1_UC2_1A\ |\}\\
M_{UC2} &= (F1_UC2(FNAME, F1_UC3_Ext_UC2_Cleanup), A_{I_{UC2}}, A_{O_{UC2}})
\end{aligned}$$

4.7. Feature model

The model for a feature is obtained by combining the feature active use cases with the feature memory. Let $aucs(f) = \{uc_1, \ldots, uc_K\}$ be the set of processes that models the active use cases of the feature f. Additionally, consider that extensions are connected with extension points. The following CSP process models the behaviour of the feature f, where it is assumed that the parameters for each use case process is a subset of that for the feature.

$$\begin{aligned}
&f(f_parameters) =\\
&\quad ((uc_1(uc_1_params) \Box \cdots \Box uc_K(uc_K_params));\ END\\
&\qquad |[\alpha_{f_MEMORY} \cup \{success\}]|\\
&\quad (f_MEMORY \vartriangle END)) \setminus (\alpha_{f_MEMORY} \cup \{success\})
\end{aligned}$$

It is the parallel composition between the choice among the processes that model the active use cases of the feature and the feature memory. Identically to a use case model, the special event $success$ and the process END are used to force the successful termination of the feature behaviour whenever an active use case does terminate. For instance, the following process models the Important Messages Feature.

$$\begin{aligned}
&F1(FNAME) =\\
&\quad ((F1_UC2(FNAME, F1_UC3_Ext_UC2_Cleanup) \Box F1_UC3)\\
&\qquad |[|\ aF1_MEM \cup \{success\}\ |]|\\
&\quad (F1_MEMORY \vartriangle END)) \setminus aF1_MEM \cup \{success\}
\end{aligned}$$

On the left-hand side of the parallelism above, the feature $F1$ is modelled as a choice between the processes that model the active use cases ($F1_UC2$ and $F1_UC3$). Note that the process $F1$ is parametrised by the folder name (FNAME).

Feature I/O process Consider that $ucs(f)$ represents the set of use cases (active or not) of the feature f. Let f be the process that models the feature f. The I/O process for this feature is

$$M_f = (f, A_{I_f}, A_{O_f})$$

where

$$\begin{aligned}
A_{I_f} &= \bigcup_{uc \in ucs(f)} A_{I_{uc}}\\
A_{O_f} &= \bigcup_{uc \in ucs(f)} A_{O_{uc}}
\end{aligned}$$

For instance, the I/O process of the feature F1 is

$$M_{F1} = (F1(FNAME), A_{I_{M_{UC1}}} \cup A_{I_{M_{UC2}}} \cup A_{I_{M_{UC3}}}, A_{O_{M_{UC1}}} \cup A_{O_{M_{UC2}}} \cup A_{O_{M_{UC3}}})$$

4.8. System model

Feature processes are composed to model the document behaviour. With the rich repertoire of CSP operators, features can be combined using any process composition operation. Here we consider a common situation in our application domain where features offer alternative services as user actions. Therefore, we model the system behaviour as the choice among all the feature processes. The extension to deal with concurrent applications is briefly discussed in the concluding section, as a topic for future work. Let $features = f_1, \ldots, f_k$ be the set of features contained in the document, and, $params(f) = p_1, \ldots, p_w$ the function that yields the list of parameters of the feature process f. The CSP model for the use case document is specified as the process.

$$S(params(f_1), \ldots, params(f_k)) = \Box f : features \bullet f$$

The behaviour of the process $S(\ldots)$ is the choice among the feature processes. This process is parametrised by the parameters of the features. For instance, the process that models the sample document presented in Fig. 4, with the singleton feature $F1$, is

$$S(F1_FNAME) = F1(F1_FNAME)$$

An instance of the document behaviour is obtained by giving concrete values to the parameters of the process S. In the Important Messages Feature, the values for `FNAME` are Inbox and Outbox. Thus, we define $nametype\ \ tF1_NAME = \{Inbox, Outbox\}$, where the names Inbox and Outbox are introduced by the datatype

$$datatype\ \ F1_NAME = Inbox \mid Outbox$$

For instance, the following process models an instantiation of the sample document.

$$System = S(Outbox)$$

System I/O process Considering that *features* represents the set of features of a document, the I/O process for the document is

$$M = (System, A_I, A_O)$$

where

$$A_I = \bigcup_{f \in features} A_{I_f}$$

$$A_O = \bigcup_{f \in features} A_{O_f}$$

For instance, the following I/O process models the sample document introduced in this chapter

$$M = (System, A_{I_{F1}}, A_{O_{F1}})$$

4.9. State based test selection

The generation and selection of tests based on specific states is similar to that presented in Sects. 3.2 and 3.3; this is an advantage of using a more abstract framework for test case generation, particularly the CSP process algebra with refinement notions, which allows us to refrain from designing explicit generation algorithms. The essential difference, nevertheless, is that now we consider a state based input model, whose structure is the composition of flow and memory processes. This section shows how we can benefit from such a structure to select test scenarios that cover specific states of the model.

Before showing how to perform test selection of particular states, we show that we can generate and select test cases from a state based model with the same approach presented in Sect. 3.2. Let $M = (System, A_I, A_O)$ be an I/O process that models a system based on the introduced feature and use case template, and let TP be a test purpose for $System$. As already said, the event mem_update is communicated by the control flow and is useful in the construction of test purposes that select particular states of the model. Consider that such an event belongs to the control alphabet of $System$, say $\alpha_{control}$, such that $\alpha_{control} \subseteq \alpha_{System}$ and $\alpha_{control} \cap (A_I \cup A_O) = \emptyset$. Consequently, this event is not part of a test scenario, and it is hidden from the process used as the input for test generation ($System$), as well in the parallel product. Thus, the parallel product between $System$ and TP (defined in Sect. 3.3) is the same as before, but extending the synchronisation set with the event mem_update:

$$PP(System, TP) = (System \mid[A_I \cup A_O \cup \alpha_{control}]\mid TP)$$

The refinement expression that can yield test scenario as counter-examples becomes

$$System \setminus \alpha_{control} \sqsubseteq_\tau PP(System, TP) \setminus \alpha_{control}$$

As an example, we can select from the process $System$ the test scenarios that match the test purpose TP_2, which differs from TP_1 (see Sect. 3.3) only in the alphabets.

$$TP_2 = UNTIL(\alpha_{System}, \{cleanUpMsgs\}, UNTIL(\alpha_{System}, \{msgMovedToIMDisp\}, ACCEPT(1)))$$

The parallel product of $System$ and the test purpose TP_2 is $PP(System, TP_2)$, which is used in the following refinement relation.

$$System \setminus \alpha_{control} \sqsubseteq_\tau PP(System, TP_2) \setminus \alpha_{control}$$

Checking this relation with FDR yields as counter-example the test scenario that follows.

$$System_TP_2_ts_1 = \langle goTo.Outbox, msgsDisp.Outbox, in_F1_UC1_2M_x.F1_Messages.\{M.1, M.0\}, selectMsgs.Outbox, msgsHighlighted.Outbox, selMoveToIMOpt, cleanUpReqDisp, out1_F1_UC2_1A.1, in_F1_UC3_1M_x.F1_Messages.\{M.2\}, selectImpMsgs, msgsHighlighted, cleanUpMsgs, cleanUpPerformed, goTo.Outbox, msgsDisp.Outbox, in_F1_UC1_2M_x.F1_Messages.\{M.0\}, selectMsgs.Outbox, msgsHighlighted.Outbox, selMoveToIMOpt, msgMovedToIMDisp, accept.1\rangle$$

This scenario is the shortest one that matches TP_2; removing the parameters, inputs and outputs it reduces to $F1_UC1_TP_1_ts_1$ (defined in Sect. 3.3).

Now we show how to benefit from the structure of the model to select particular states. This can be performed by allowing the test purpose process to access the variable values that are communicated by the memory processes through get events. If such events were not hidden in the composition of memory and use case processes they would be offered by the process $System$, and the test purpose process, which is composed with the $System$ process in the parallel product, could synchronise on them to read variable values, identically to the way processes representing use case flows access and update variable values in the memory. However, due to the synchronous communication semantics of CSP, the synchronisation of the test purpose in get events would obligate the control flow and memory processes (in the $System$ process) to synchronise in get events whenever the test purpose did, and vice-versa. As a consequence, the parallel composition of control flow, memory and test purpose processes would block whenever the three processes were not ready to get involved in a multisynchronisation. To avoid such a blocking, we use independent events, named get', for the test purpose to access variable values. Differently from get events, get' events are not hidden and are offered by the $System$ process. The get' events allow the test purpose to read variables independently of the control flow processes. Figure 5 depicts the structure of the process $System$ with get' events. The mem_update event is also used by the test purpose to select specific states, so it is also visible.

Let MEM be a memory process. We clone the get events (and keep the set events) of MEM with the expression $MEM[\![^{get.*\,,\, get.*} / _{get.*\,,\, get'.*}]\!]$, where $.*$ represents the variable values. Such a renaming keeps the get events (renaming it by itself) and creates a copy (get'). As already shown, the get events are hidden in the composition of memory with use case processes, however the copies remain visible (get'). For instance, the process $F1_MEM$ is replaced by $F1_MEM[\![^{get.*\,,\, get.*} / _{get.*\,,\, get'.*}]\!]$ in the definition of the process $F1$. As a consequence of the hiding in the expression of $F1$, the events $get.F1_Messages.v$ become internal to the process $F1$ and the events $get'.F1_Messages.v$ remain visible.

From this point on assume that every memory process is renamed to communicate get' events. In our example, the processes $F1_UC3_MEM$ and $F1_MEM$ are replaced by their renamed versions. As a consequence of such a renaming, get' events, as well as the mem_update event, also belong to the control alphabet of the process $System$ ($\alpha_{control}$). Once get' events belong to α_{System}, they can be used to construct CSP test purposes for $System$ (see Definition 2). Using the control alphabet a test purpose process can match the states of interest, in addition to input and output events, and communicate the mark event when the combination of state and traces is reached.

The following process is an example of a test purpose that can read variable values to match specific states of $System$.

$$
\begin{aligned}
TP_3 = & \ get'!F1_selected?selected \rightarrow \\
& \quad get'!F1_important?important \rightarrow \\
& \quad if((\#(value(selected)) + \#(value(important))) == F1_MAX) \ then \\
& \qquad UNTIL(aS, \{msgMovedToIMDisp\}, ACCEPT(1)) \\
& \quad else \\
& \qquad UNTIL(aS, \{mem_update\}, TP3)
\end{aligned}
$$

Initially, the test purpose TP_3 reads from the feature memory ($F1_MEMORY$) two variables: the set of selected messages (`selected`) and the set of messages in the Important Messages Folder (`important`). If the cardinality of selected messages plus the cardinality of the important messages equals the maximum capacity of the folder ($F1_MAX$), the test purpose continues the selection until it finds the system response that indicates selected messages are moved ($msgMovedToIMDisp$), and communicates a mark event. Otherwise, if the sum is different, the test purpose recurses just after a memory update is found.

According to the new definition for the parallel product presented in this section, mem_update and get' events belong to the synchronisation set of the parallel composition of the model and TP_3. If the test purpose TP_3 recursed without looking for a memory update, the input model would not progress in the parallel product, because the test purpose would offer only get' events, which block the communication of other events of the model process. Consequently, the test purpose would read the variables only in the states of the model reached after the empty trace. This is the main reason why we need the mem_update event: it allows the input model to progress to a new memory state and notify the test purpose of such a change. This is not particular to the test purpose $TP3$; in any test purpose that looks for particular states, the mem_update event is looked for ($UNTIL(aS, \{mem_update\}, \ldots)$ whenever the test to assess some state fails and the test purpose recurses to search again for the same state.

The parallel product of $System$ and $TP3$ is $PP(System, TP_3)$; it is used to select test scenarios in the refinement relation.

$$System \setminus \alpha_{control} \sqsubseteq_{\tau} PP(System, TP_3) \setminus \alpha_{control}$$

The verification of the above refinement with FDR yields the following counter-example.

$$
\begin{aligned}
& System_TP_3_ts_1 = \langle goTo.Outbox, msgsDisp.Outbox, in_F1_UC1_2M_x.F1_Messages.\{M.0\}, \\
& selectMsgs.Outbox, msgsHighlighted.Outbox, selMoveToIMOpt, msgMovedToIMDisp, accept.1\rangle
\end{aligned}
$$

This scenario describes the situation where the user goes to the Outbox folder and selects one message to be moved to the Important Messages Folder. Since the sum of the selected set with the current number of messages on the Important Messages Folder does not exceed the capacity of the target folder (in fact is the maximum number allowed to move), the message is moved.

Using get' events it is also possible to read variables from local memories (as, for instance, the variable F1_UC3_selected), in the same way the feature memory is accessed.

4.10. Test scenario final state

In addition to the selection of test scenarios that match particular states, it is also possible to trace the variable states in the specification after performing test scenarios. Such an information is very useful to define the system state after the test execution. Initial and final states can be used to document the pre and post states of the generated tests. A possible usage of pre and post states is to find the optimal test execution ordering to reduce the setup cost during test execution. In [LISA09] we introduce an algorithm to order a set of test cases based on their pre and post states.

The post state of a test scenario can be easily achieved by modifying the process expression for the test purpose and that for the refinement verification. The test purpose is modified to assess the variables in the features scope after the last event of the test scenario is found. The variable values represent the test final conditions. Moreover, the expression for the refinement verification is modified (both on the left and on the right-hand sides) to hide only mem_update events, keeping visible get' events. As a consequence, the counter-examples yielded by the verification will exhibit get' events, in addition to input and output events. In fact, since get' events are control

events they are not part of the test scenario, so they are removed from the counter-example traces. However, since the get' events after the last event represent final conditions they are used to document the test final conditions, whereas the other occurrences of get' are simply discarded.

We illustrate this with the test purpose TP_3', a modified version of TP_3.

$$
\begin{aligned}
TP_3' = {} & get'!F1_selected?selected \rightarrow \\
& get'!F1_important?important \rightarrow \\
& if((\#(value(selected)) + \#(value(important))) == F1_MAX)\ then \\
& \quad UNTIL(aS, \{msgMovedToIMDisp\}, TP_3'_END(1)) \\
& else \\
& \quad UNTIL(aS, \{mem_update\}, TP_3')
\end{aligned}
$$

The main difference between TP_3 and TP_3' is that the latter replaces $ACCEPT$ by $TP_3'_END$, which we denominate test purpose finalisation. Basically, the behaviour of $TP_3'_END$ is to look for memory updates, read the specification variables and behave as $ACCEPT$. The specification for $TP_3'_END$ is as follows.

$$
\begin{aligned}
TP_3'_END(id) = UNTIL(aS, mem_update, \\
& get'!F1_folder?v0 \rightarrow \\
& get'!F1_selected?v1 \rightarrow \\
& get!F1_important?v2 \rightarrow \\
& ACCEPT(id))
\end{aligned}
$$

If the last event of the generated test scenarios is followed by an update event (set), the initial behaviour of the test purpose finalisation is to look for the mem_update event. It is demanded to assure that the effects produced by the test scenario are accomplished before the variables are traced. This is the case of the process $TP_3'_END$, which looks for mem_update because the effects of moving a message ($msgMovedToIMDisp$) are to remove messages from a source folder and put them in the hot messages folder. By synchronising on that event the immediate memory updates are taken (see Variables Update in Sect. 4.6). Then, this process assesses the three feature variables and behaves as $ACCEPT(1)$.

The refinement expression that generates tests and reads the final state of the test scenarios is the following.

$$System \setminus \{mem_update\} \sqsubseteq_{\tau} PP(System, TP_3') \setminus \{mem_update\}$$

The verification of the above refinement with FDR yields the following counter-example.

$$
\begin{aligned}
& System_TP_3'_ts_1 = \langle get'.F1_selected.F1_Messages.\{\}, get'.F1_important.F1_Messages.\{M.2\}, \\
& goTo.Outbox, msgsDisp.Outbox, in_F1_UC1_2M_x.F1_Messages.\{M.0\}, selectMsgs.Outbox, \\
& msgsHighlighted.Outbox, get'.F1_selected.F1_Messages.\{M.0\}, get'.F1_important.F1_Messages.\{M.2\}, \\
& selMoveToIMOpt, msgMovedToIMDisp, get'.F1_folder.F1_Messages.\{M.1\}, \\
& get'.F1_selected.F1_Messages.\{M.0\}, get'.F1_important.F1_Messages.\{M.2, M.0\}, accept.1\rangle
\end{aligned}
$$

Let $filter(s, X)$ be a function that filters a trace s by removing the events from the set X and keeping the others. Formally, $filter(s, X) = s \upharpoonright \Sigma - X$. Removing the control and mark events from the above trace we have exactly $System_TP_3_ts_1 = filter(System_TP_3'_ts_1, \alpha_{control} \cup \{accept.1\})$, which is the test scenario we are interested on. Furthermore, the last three occurrences of get' in $System_TP_3'_ts_1$ trace the final state after executing $System_TP_3_ts_1$. That is, the source folder has a unique messages $M.1$, the moved message is $M.1$ and the important messages folder keeps the set of messages $\{M.2, M.0\}$.

Potentially, the state based test selection approach presented in this section can be applied to select tests in CSP models that represent other kinds of systems and have a similar structure to that of our memory process.

A potential advantage of our testing theory entirely based on a process algebra is abstraction, modularity and reuse. As illustrated in this section, the same test case generation strategy, with only alphabet adaptations, could be used to generate and select tests both for control flow processes as well as for state based ones. This contrasts with approaches based on more operational models, such as those for LTS [JJ05, CJRZ01] where specific, and substantially distinct algorithms, are developed for control and state based testing.

5. Sound test cases

In conformance testing, a basic 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, which is another important contribution of our work.

CSP Input–Output Conformance To address soundness, conformance testing [Tre99] requires the definition of an implementation relation between the domain of specifications and that of implementations. In our work, elements of such domains are expressed as I/O processes. Thus, to present our definition for such a relation we assume as *test hypothesis* [CG07] that there is an I/O process which specifies an implementation under test, say IUT. From now on, we consider the general model for an implementation the I/O process IUT defined as the tuple $(P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$, and the general model for an specification the I/O process S defined as (P_S, A_{I_S}, A_{O_S}).

We also assume that implementations are always able to accept any input from the alphabet (input enabled) and always produce some output after some input (output enabled). In practice, it is not common to have input enabled implementations. Nevertheless, since implementations are stimulated through some test driver (test script in automatic execution or by a tester in manual execution) input enabledness can be achieved by considering the combination of the implementation and the test driver, which discards the input events that are not enabled on the implementation. In our domain of mobile applications the GUI constantly displays outputs, even if no input is performed, so output enabledness is a natural property of this kind of application.

Input enabledness and output enabledness are formalised by the two following definitions. An I/O process is input enabled when the inputs communicated after each of its traces is the same as its input alphabet. Consider that the function $initials(P) = \{a \mid \langle a \rangle \in \mathcal{T}(P)\}$ yields the set of events offered by the process P. Below is the formal definition of an input enabled I/O process.

Definition 3 (Input enabled I/O process). Let $M = (P_M, A_I, A_O)$ be an I/O process. Then, M is input enabled iff $\forall\, t : \mathcal{T}(P_M) \bullet A_I \subseteq initials(P_M/t)$

An implementation is output enabled when we can always find an output event. Below is the formal definition of an output enabled I/O process.

Definition 4 (Output enabled I/O process). Let $M = (P_M, A_I, A_O)$ be an I/O process. It is output enabled iff $\forall\, t : \mathcal{T}(P_M);\ \exists\, i : A_I,\, o : A_O \bullet o \in initials(P_M/t \frown \langle i \rangle)$

Our implementation relation **cspio** (CSP Input–Output Conformance), formalised in Definition 5, is the basis for our generation of sound CSP test cases. Consider that the function $out(M, s)$ gives the set of output events of the process component of the I/O process M, P_M, after the trace s. Formally, $out(M, s) =\ if\ \ s \in \mathcal{T}(P_M)\ \ then\ \ initials(P_M/s) \cap A_{O_M}\ \ else\ \ \emptyset$. The relation **cspio** establishes that any output event observed in an implementation model IUT is also observed in the specification S, after any trace of S. In this case, IUT **cspio** S.

Definition 5 (CSP input–output conformance). Consider $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ an implementation model and $S = (P_S, A_{I_S}, A_{O_S})$ a specification, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. Then

$$IUT\ \ \mathbf{cspio}\ \ S \equiv \forall\, s : \mathcal{T}(P_S) \bullet out(IUT, s) \subseteq out(S, s)$$

Theorem 5.1 below captures **cspio** using process refinement. The proof is presented in Appendix A.

Theorem 5.1 (Verification of cspio). Let $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ be an implementation model, and $S = (P_S, A_{I_S}, A_{O_S})$ a specification, with $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. Then IUT **cspio** S holds iff the following refinement holds.

$$P_S\ \sqsubseteq_\tau\ (P_S \bigtriangleup ANY(A_{O_{IUT}}, Stop))\ \ |[A_{I_{IUT}} \cup A_{O_{IUT}}]|\ \ P_{IUT}$$

The intuition for this theorem is as follows. First we give the intuition why new inputs on P_{IUT} are allowed by the above expression. Consider an input event that occurs in P_{IUT}, but not in P_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 not recorded in the traces model, such refused events are not in the traces of the right-hand side process; So, new P_{IUT} inputs are allowed by the above refinement, as such events will not belong to the traces of the left- nor of the right-hand side process of the above expression. Now we give the intuition why new outputs (for a common trace) are not allowed. The objective of the interruption with the process $ANY(A_{O_{IUT}}, Stop)$ is to

avoid that the right-hand side process refuses output events that the implementation can perform but P_S cannot. Thus, $ANY(A_{O_{IUT}}, Stop)$ allows that such outputs be communicated to P_{IUT}. Finally, if P_{IUT} 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 P_{IUT} generating deadlock from the trace where the input has occurred, in such a way that any event that comes after is not part of the traces. Also, it keeps in the traces all the output events of P_{IUT} for common traces with P_S, allowing a comparison in the traces model.

When detailing the proof of Theorem 5.1 (as presented in Appendix A.1) we have uncovered a subtle technicality with respect to a previous characterisation of the Theorem [NSM08], which used interleaving in place of interruption on the right-hand side process. The expression with interleaving let the implementation synchronise with an output offered by the process $ANY(A_{O_{IUT}}, Stop)$ and the specification process to progress after that. This makes the refinement expression not to hold in particular situations where the implementation communicates new inputs after a common prefix of the specification. Obviously, new implementation inputs are allowed by **cspio**, thus it is a false counter-example. The change from interleaving to the interruption operator removed such an issue.

As an example, consider the I/O process $S1 = (P_{S1}, \{i1\}, \{o1\})$ is a sample specification, and the I/O processes $IUT1 = (P_{IUT1}, \{i1\}, \{o1, o2\})$ and $IUT2 = (P_{IUT2}, \{i1\}, \{o1, o2\})$ are candidate implementations (input enabled and output enabled). Below are the specifications for P_{S1}, P_{IUT1} and P_{IUT2}.

$$P_{S1} = i1 \rightarrow o1 \rightarrow P_{S1}$$

$$\begin{aligned} P_{IUT1} = i1 \rightarrow (&o1 \rightarrow P_{IUT1} \\ &\Box \\ &i1 \rightarrow (o2 \rightarrow P_{IUT1} \\ &\qquad\quad \Box \\ &\qquad\quad P_{IUT1})) \end{aligned}$$

$$\begin{aligned} P_{IUT2} = i1 \rightarrow (&o1 \rightarrow P_{IUT2} \\ &\Box o2 \rightarrow P_{IUT2} \\ &\Box P_{IUT2}) \end{aligned}$$

We can mechanically verify, using a tool such as FDR, the expressions $IUT1$ **cspio** $S1$ and $IUT2$ **cspio** $S1$ using the refinement in Theorem 5.1. One then finds out that the first relation holds and the second does not. Despite the fact that the implementation $IUT1$ performs an output ($o2$) that is not in the specification $S1$, such an output happens after traces that are not in $\mathcal{T}(S1)$, for instance, the trace $\langle i1, i1\rangle$. According to Definition 5, any behaviour of $IUT1$ after new traces is allowed. However, because $out(IUT2, \langle i1\rangle) = \{o1, o2\}$ is not a subset of $out(S1, \langle i1\rangle) = \{o1\}$, the trace $\langle i1\rangle$ is a counter-example for $IUT2$ **cspio** $S1$.

In practice, if we know IUT we can verify IUT **cspio** S by checking the relation in Theorem 5.1 directly, as illustrated above. This is equivalent to generating all the traces of S and exercising them against the implementation according to **cspio**. However, normally we do not know IUT and the number of traces of S is infinite. Therefore, we need to exercise the implementation with a selected subset of test cases and look for possible violations of IUT **cspio** S during the test execution.

Test Case and Soundness Before defining a sound test case, we need to state what is the meaning of a test case and its execution.

A test case, say TC, is an I/O process generated from a specification S which interacts with an implementation IUT to indicate whether the implementation conforms to the specification according to **cspio**. Test outputs stimulate the implementation, and the implementation responses stimulate the test, so the test and implementation alphabets are inverted to model the opposite directions in the communication. Formally, let S be the model for the specification and IUT be that for the implementation, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. A test case $TC = (P_{TC}, A_{I_{TC}}, A_{O_{TC}} \cup VER)$ generated from S to test IUT is an I/O process, which inputs events from $A_{I_{TC}} \subseteq A_{O_{IUT}}$, and outputs events from $A_{O_{TC}} \subseteq A_{I_S} \cup VER$, with $VER = \{pass, fail, inc\}$, such that $VER \cap (A_{I_{IUT}} \cup A_{O_{IUT}}) = \emptyset$. The constraints over test case alphabets follow directly from the definition for **cspio** (Definition 5) and alphabet inversion: test inputs are contained in the implementation outputs because

implementations can produce more outputs than the specification; and a test case outputs only specification inputs because a test case is obtained from the specification alphabet.

The execution of a test TC against an implementation IUT, say $EX(IUT, TC)$, is captured by the parallel composition $P_{IUT} \mid [A_{I_{IUT}} \cup A_{O_{IUT}}] \mid P_{TC}$. Such an execution can yield a verdict event communicated by the test case, which behaves as one of the following processes: $PASS = pass \rightarrow Stop$ to express when the test passes in the execution, $INC = inc \rightarrow Stop$ for an inconclusive execution, and $FAIL = fail \rightarrow Stop$ for a failed execution.

Let $EXEC$ be the CSP model for a test execution. The presence of a verdict event $v \in VER$ in the traces of the test execution $EXEC$ can be easily verified with the refinement.

$$EXEC \setminus (A_{I_{IUT}} \cup A_{O_{IUT}}) \sqsubseteq_{\tau} v \rightarrow Stop$$

On the left-hand side of the above expression, input and output events are hidden from the execution process, so the unique events communicated are verdicts. Consequently, if the refinement holds, the trace $\langle v \rangle$ belongs to the traces of the execution and v is a possible verdict; otherwise, if it does not hold, the trace $\langle v \rangle$ is not in the traces of the execution, so the verdict v never happens.

As an example of such a verification, we define the test case $TC_0 = (P_{TC_0}, \{o1, o2\}, \{i1\})$ generated from $S1$ to test $IUT1$, such that $P_{TC_0} = i1 \rightarrow (o1 \rightarrow PASS \Box o2 \rightarrow FAIL)$. Let $EXEC1 = EX(IUT1, TC_0)$ be the test execution of TC_0 against $IUT1$. Using the above refinement we have the expression $EXEC1 \setminus \{i1, o1, o2\} \sqsubseteq_{\tau} pass \rightarrow Stop$, which can be checked with FDR to confirm that $pass$ is a possible verdict for the execution of TC_0 against $IUT1$. However, $fail$ is not a possible verdict. This can be confirmed verifying the expression $EXEC1 \setminus \{i1, o1, o2\} \sqsubseteq_{\tau} fails \rightarrow Stop$, which does not hold. On the other hand, the execution of TC_0 against $IUT2$ can lead to a fail. This can be confirmed verifying that the expression $EX(TC_0, IUT2) \setminus \{i1, o1, o2\} \sqsubseteq_{\tau} fail \rightarrow Stop$ holds.

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 suite is sound if all its tests are also sound. As already seen, a CSP test execution of a test TC with an implementation IUT fails when the test execution $EX(IUT, TC)$ has the event $fail$ as part of at least one of its traces. A formalisation of soundness is as follows.

Definition 6 (**Sound test case**). Let IUT be an implementation I/O process, S the specification, TC a test case I/O process and $EXEC = EX(IUT, TC)$ the execution of TC against IUT. Then TC is a sound test case if the following holds.

$$\langle fail \rangle \in \mathcal{T}(EXEC \setminus (A_{I_{IUT}} \cup A_{O_{IUT}})) \Rightarrow \neg(IUT \ \mathbf{cspio} \ S)$$

In other words, a sound test case does not generate false fails. For instance, the test case TC_0 is a sound test case, because it does not fail when run against a valid implementation according to **cspio** (for instance, $IUT1$). Furthermore, if the same test case fails (for instance, when it runs against $IUT2$, as discussed previously), we are sure the implementation under test does not conform to the specification $S1$, according to **cspio**.

Constructing Sound Test Cases We now show how to construct the process component (P_{TC}) of a test case TC from a test scenario ts generated from a specification S to test IUT. The resulting test case is able to detect invalid implementations according to **cspio**. First, we create an annotated trace ($atrace$) obtained from ts that records the output events offered by the specification in the point each test scenario event is offered. Formally, $atrace = \langle (ev_1, outs_1), \ldots, (ev_{\#ts}, outs_{\#ts}) \rangle$, where ev_i is the ith element of ts and $outs_i$ is the set of output events (different from ev_i) after the specification performs the trace $\langle ev_1, \ldots, ev_{i-1} \rangle$, such that $outs_i = if (ev_i \in A_O)\, then\, (out(S, \langle ev_1, \ldots, ev_{i-1} \rangle) - \{ev_i\})\, else\, \emptyset$, for $1 \leq i \leq \#ts$ and $\langle ev_1, \ldots, ev_{i-1} \rangle$ belongs to the prefixes of ts.

The function $TC_BUILDER(atrace)$ yields the process component of a sound test case constructed from an annotated trace $atrace$.

$$TC_BUILDER(\langle\rangle) = PASS$$

$$TC_BUILDER(\langle (ev, outs) \rangle ^\frown tail) = SUBTC((ev, outs)); \ TC_BUILDER(tail)$$

where

$$SUBTC((ev, outs)) = ev \rightarrow Skip \, \Box \, (ev \in A_{O_S} \ \& \ ANY(outs - \{ev\}, INC) \, \Box \, ANY(A_{O_{IUT}} - outs, FAIL))$$

The process $TC_BUILDER(atrace)$ recursively behaves like the process $SUBTC$ for each pair $(ev, outs)$ of $atrace$ and yields the process $PASS$ when the last element of $atrace$ is reached. The goal of the process $SUBTC$ is to create the body of the test. Its primary behaviour is to offer the event ev to the implementation and to end with success ($Skip$) after communicating it. If ev is a test output (implementation input), it is communicated to the implementation and the test fragment ends with success—the implementation is always ready to accept inputs (input enabledness). However, if ev is a test input (implementation output), the process must be ready to synchronise with any output response of P_{IUT}—the test cannot block implementation outputs. If the P_{IUT} response matches ev, the test synchronises on this event and finishes with success. Otherwise, if the implementation output does not match ev, the test must be ready to accept this output and decide whether the execution fails or is inconclusive. In case the P_{IUT} communicates an event that belongs to $outs - \{ev\}$, the test reaches the verdict inconclusive since the P_{IUT} response is not exactly the one expected by the test scenario (ev), but it is an output produced by the specification. Otherwise, if the P_{IUT} communicates an output event not in the specification, which means that the event belongs to $A_{O_{IUT}} - outs$, the implementation does not **cspio** conform to the specification, so the test reaches the verdict fail.

Theorem 5.2 (TC_BUILDER is sound). Let $S = (P_S, A_{I_S}, A_{O_S})$ be a specification, ts a test scenario from S and $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ an implementation model, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. If $atrace$ is an annotated trace obtained from ts, then $TC = (TC_BUILDER(atrace), A_{I_{TC}}, A_{O_{TC}})$ is a sound test case, such that $A_{I_{TC}} = A_{O_{IUT}}$ and $A_{O_{TC}} = A_{I_S}$.

The proof of Theorem 5.2 is presented in Appendix A. To exemplify a sound test case, we use a prefix of the test scenario $System_TP_2_ts_1$ extracted from the specification $M = (System, A_I, A_O)$ to test an implementation with the same alphabet of M and build the process $TC_1 = TC_BUILDER\ (atrace_ts_1)$, where $atrace_ts_1 = \langle(goTo.Outbox, \emptyset), (msgsDisp.Outbox, \emptyset), (in_F1_UC1_2M_x.F1_Messages.\{M.1, M.0\}, \emptyset), (selectMsgs.Outbox, \emptyset), (msgsHighlighted.Outbox, \emptyset), (selMoveToIMOpt, \emptyset), (cleanUpReqDisp, \emptyset), (out1_F1_UC2_1A.1, \emptyset)\rangle$. The resulting process is

$$
\begin{aligned}
&TC_1 = goTo.Outbox \rightarrow Skip;\\[6pt]
&(msgsDisp.Outbox \rightarrow Skip\\
&\Box\ ANY(A_O - \{msgsDisp.Outbox\}, FAIL));\\[6pt]
&in_F1_UC1_2M_x.F1_Messages.\{M.1, M.0\} \rightarrow Skip;\\
&selectMsgs.Outbox \rightarrow Skip;\\[6pt]
&(msgsHighlighted.Outbox \rightarrow Skip\\
&\Box\ ANY(A_O - \{msgsHighlighted.Outbox\}, FAIL));\\[6pt]
&selMoveToIMOpt \rightarrow Skip;\\[6pt]
&(cleanUpReqDisp \rightarrow Skip\\
&\Box\ ANY(A_O - \{cleanUpReqDisp\}, FAIL));\\[6pt]
&(out1_F1_UC2_1A.1 \rightarrow PASS\\
&\Box\ ANY(A_O - \{out1_F1_UC2_1A.1\}, FAIL));
\end{aligned}
$$

The process TC_1 tests whether the implementation performs the scenario in which the user tries to move messages from the Outbox to the Important Messages Folder without success, due to space limitation in the target folder. The test passes if the whole test scenario is observed in the test execution, and it fails if the implementation produces some output that is not in the test scenario and is not foreseen by the specification (for instance, the implementation response $cleanUpReqDisp$ after the test output $msgsDisp.Outbox$).

Test Case ID	Case Description	Procedure	Expected Results
F1_1	Objective:	Moving messages from Outbox to Important Messages when the storage is full.	
	Initial Conditions:	folder = {M.0,M.1}, selected = {}, important = {M.2}	
	Step Number		
	1	Go to folder. (Outbox)	All Messages are displayed. (Outbox)
	2	Select message(s). ({M.1,M.0})	Message(s) are highlighted.
	3	Select "Move to Important Messages" option.	"Message storage has not enough space" is displayed. "Clean Up request" is displayed for X messages. (1)
	Final Conditions:	folder = {M.0, M.1} selected = {M.0,M.1} important = {M.2}	

Fig. 13. CNL Test case for TC_1

There is no inconclusive verdict in the test TC_1 because there are no alternative outputs in the process specification $System$ for the test scenario $System_TP_2_ts_1$. As an example of an inconclusive verdict, consider the atraces for a prefix of the test scenario $F1_UC1_TP_1_ts_1$ to test an implementation with the same alphabet of the I/O process F introduced in Sect. 3.1, say $atraces' = \langle (goTo, \emptyset), (msgsDisp, \emptyset), (selectMsgs, \emptyset), (msgsHighlighted, \emptyset), (selMoveToIMOpt, \emptyset), (cleanUpReqDisp, \{msgMovedToIMDisp\}) \rangle$. Due to the non-deterministic behaviour of the specification process $F1_UC1$, the implementation is permitted to produce any output of the set $\{cleanUpReqDisp, msgMovedToIMDisp\}$ after the user selects to move a message ($selMoveToIMOpt$). Hence, the test case process $TC_2 = (TC_BUILDER(atraces'), A_{I_F}, A_{O_F})$ leads to an inconclusive behaviour if the implementation produces an output event $msgMovedToIMDisp$, while the test waits for the output $cleanUpReqDisp$.

According to Theorem 5.2 TC_1 is a sound test case. To illustrate the last phase of the automation workflow presented in Sect. 2 (Fig. 1), we present the CNL representation for the sound test case TC_1, presented in Fig. 13 in a tabular representation that is suitable for manual test execution. It encloses the id, objective, initial conditions, steps and final conditions for the test case. It is a simplification of the format used in the BTC project, which includes additional fields. The objective field for TC_1 can be automatically generated by including a description for each use case flow. The TaRGeT tool concatenates the descriptions for the use cases covered by the test scenario and outputs the content for the test objective. Similarly, the tool can produce the content for the initial conditions by concatenating the content of the `'System State'` field for the use case steps covered by the test scenario. In Fig. 13, the initial conditions are generated from the initial value for the variables. The steps 1, 2 and 3 were obtained from the mapping of of CSP events back into CNL sentences. The test final conditions are filled with the information obtained from the test scenario final state.

6. Related work

Test Generation from Use Cases Many test generation approaches that input some form of use cases have been proposed. Many approaches adopted graphical representation for the use cases [NS11, NFLTJ06, RG99, BL02, WP99]. For instance, [WP99] describes a method, based on equivalence partition for testing multi-panel systems (systems with a form based interface). Control flow is specified by a graph that models the possible transitions between use cases. Test generation consists in finding the sequence of use cases and respective input values that are able to cover the structure of the model. In this approach the test model represents only input data and the navigation flows through the forms. Differently from our approach, system outputs are not explicitly represented and there is no explicit notion of state.

The approaches more related to ours are those that output textual test cases to test the system through the GUI [HVFR05, Goi10, BG03, SC08]. The work in [HVFR05] inputs UML activity diagrams that represent textual use cases and outputs textual test cases to be executed against the system interface. In the diagram, stereotypes are used to distinguish input actions from system responses, and to associate data to input actions; also, sub-diagrams can be included. Input data and guards are specified using TSL (test specification language). Test generation consists in visiting the transitions of a directed graph extracted from the activity diagram. By default, all transitions and data partitions are covered by the generated tests, but the user can prioritize the activities and data to be covered. Nevertheless, activity diagrams can be partially obtained from textual use cases, the test model is the activity diagram, not a use case specification. In addition, such a model does not allow the specification of variables nor output values, and the test postconditions are not automatically generated.

The author in [Goi10] introduces TSD (Test Script Diagram), a diagrammatic representation for use cases, which allows the specification of flows, loops, input data and sub-diagrams. Types are defined as equivalence classes to be associated with the input variables. Textual use cases are semi-automatically translated to TSD and tests are automatically generated from TSD diagrams. The generated tests are suitable for manual test execution. The selection of tests is based on the priority of the elements, which is assigned to each element during modelling. Unlike our approach, [Goi10] does not specify state variables, and the only type is the enumeration of values.

The works reported in [BG03, SC08, SC08] use natural language to specify use cases. The method presented in [BG03] derives test scenarios from requirement structures as product line use cases (PLUCs), written in natural language and designed to express product line variabilities. Use cases are parametrised with tags; the instantiation of tags to values is described with constraints. Environment conditions and inputs are split into categories (as in the category partition method [OB88]), whose values are defined within a test specification. A test scenario is the instantiation of each tag of the use case combining the constraints and the test specification. In opposition to our approach, use case flows do not have a precise semantics.

The approach described in [SC08] is closely related to ours, since it uses natural language for the specification of use cases, maps use cases to a formal model (FSM) and generates textual test cases. In this approach the application domain is specified by the user with UML and the steps are written using a concrete syntax that refers to domain concepts and attributes, according to the format introduced in [Som06]. Use cases are translated to FSM and test scenarios are generated by traversing the FSM based on structural coverage constraints. Similarly to our use case template, the use case format follows a well defined syntax and allows the specification of step flows, conditions, operations and use case relationships. On the other hand, there is no notion of input, parameter or output, nor a test selection criteria based on specific states of the underlying model. Additionally, data is restricted to basic types (natural, boolean and enumeration), while in our approach user defined data types are allowed.

Another similar approach concerning both the input model and the test model generation is [CT08], which presents a tool that assists the specification of requirements with the purpose of generating test cases using our test generation approach. Objects and interfaces to be tested are introduced textually as requirements (formed of sentences) and used to define user scenarios (flows) that are described as use cases. Requirements and use cases are written in a particular CNL, which defines structure and syntax of requirements and use cases. Through meta-model transformations, requirements and scenarios are translated to CSP processes that are used as the input for our test generation approach. Excluding the CNL and the concepts of objects and interfaces, the use case format is the same as that which we have adopted. Nonetheless, the use case description supports only control flows, lacking data and parameters; as a consequence, the test generation cannot select specific scenario states of the model.

In summary, none of the existing approaches consider a natural language representation that mixes control and state representation, which can be used to select particular scenarios during test generation. Furthermore, none of the cited works addresses the formal properties of the generated tests like soundness of generated test cases.

Input–Output Models and Conformance Tretmans [Tre96, TB99] 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 quiescent behaviours, that we currently forbid by assuming that implementations are both input enabled and output available. As already explained, this assumption is reasonable in our application domain. On the other hand, Theorem 5.1 allows provably correct conformance verification using FDR. For **ioco**, there is an algorithm [WW08] to check conformance, but its soundness has not been addressed.

Jard et al. [JJ05] present the TGV tool that is able to select test cases based on test purposes. Clark et al. [CJRZ01] present the STG tool that generates symbolic test cases [RBJ00] based on the combination of symbolic specification and test purpose. Like TGV and STG, ATG uses test purpose as the selection criteria for test generation. However, ATG inputs CSP models, while those tools input LTS based models; TGV inputs IOLTS (Input–Output LTS) models whereas STG inputs IOSTS (Input–Output Symbolic Transition Systems) models. Both IOSTS and CSP state based models can specify inputs and outputs as well as define and manipulate data. Furthermore, both models are suitable for test selection based on specific states, and generate test cases with data. However, in contrast to our approach, which generates tests using refinement verifications on processes, the approaches in [JJ05, CJRZ01] manipulate a concrete model and define explicit algorithm to generate test cases.

Spec Explorer [VCG+08] is a model based testing tool for object-oriented reactive systems that inputs Abstract State Machines (ASM) models written in the Spec# notation (model programs) to automatically generate test cases for components developed in the Microsoft .NET framework. Model programs are symbolic partial descriptions of the system behaviour (potentially infinite) split into input and output method invocations. The test selection is based on state space exploration that is optimised by both automatic and user oriented techniques that restrict the number of states to be considered. Spec Explorer is a mature tool with a lot of documentation and support, although the MBT approach of Spec Explorer was designed for white-box testing, while our approach aims at black-box testing. Moreover, the testing theory of Spec Explorer is based on a different conformance relation that is named alternating refinement [AHKV98].

Test Generation Based on CSP A common characteristic of the existing approaches for test generation based on CSP [PS97, Sch99, CG07] is that the adopted conformance relations do not distinguish input and output events. Furthermore, test cases are not generated to verify particular scenarios of the specification, so test purposes are not considered. Peleska and Siegel [PS97] introduce a methodology for specification, design and verification of fault-tolerant systems that allows different formal methods to be combined. Based on Henessys's testing theory [Hen88], they proposed a set of conformance relations that can be characterised as CSP refinement relations and be mechanised with the FDR tool. Alternatively to mechanised conformance verification, they defined a (possibly infinite) set of test cases whose successful execution against the implementation corresponds to a proof of conformance. Cavalcanti and Gaudel [CG07] state the testability hypothesis for CSP and characterise the set of complete test cases with respect to their implementation relation that is based on traces and failures refinement of CSP. The execution of the test cases aims at showing the implementation is a valid CSP refinement for the specification. Mechanisation of the test cases is not addressed. Scheneider [Sch99] defines a partition that classifies refusable and nonrefusable events, and high-level and low-level events, for the purposes of specifying fault-tolerance systems with CSP. The focus is on the characterisation of testing relations that are able to show equivalence of processes based on the observation of events. The relations can be mechanically verified with refinement, but no approach for test generation is proposed.

Representation of State in CSP It is well known that CSP does not have variables despite they can be simulated using recursive, parametrised, processes and parallel composition [Ros98]. For instance, Roscoe [Ros11] introduces a compiler that inputs a program coded in a simple shared variable language and outputs CSP processes representing the program; generated processes are analysed with FDR to verify properties of the program execution (for instance, deadlock freedom). Basically, each program (thread) is translated into a corresponding CSP process that represents the control flow, and each variable in the program (possibly shared among threads) is translated into a process that keeps the variable state. The processes that model thread control flows and the ones that represent variable states are composed in parallel and synchronise on special channels that enable the threads to read from and write to variables. This is very similar to our composition of processes that model use cases and memories. However, in [Ros11] the variables are restricted to integers and booleans, whereas the types of the use case variables are defined by the user. Moreover, our memory processes represent the variables in a more concise manner dealing with the different types in a uniform way. This avoids the creation of a particular channel for each type, as is done in [Ros11].

Colvil et al. [CH09] introduce an extension of CSP to allow state based behaviour named CSP_σ. This enables the declaration of local and shared variables, which can be nested in different levels (hierarchical state) and includes constructs for testing and updating the variables. Since the state of our use cases is hierarchically constructed considering two levels of scope (use case and feature), we could use this approach to model the use case documents as well. However, the traces semantics of such an extension is not yet defined, thus, so far, one cannot mechanically verify traces refinements in CSP_σ, which is the basis for our test generation strategy. On the other

hand, as stated in [CH09], for any specification written in CSP_σ there is an equivalent on in CSP. From this point of view, the composition of control flow and memory processes can be seen as a pattern in CSP for modelling the access and the update of variables declared in two levels (but could be easily generalised to an arbitrary hierarchy.)

PAT [SLD08] is a CSP model checker that inputs CSP# [SLDC09], a specification language that combines process algebraic expressions with variable declarations. It is optimised for the analysis of processes that share common variables and allows LTL model checking [EEC+04] in addition to process refinement checking. Differently from CSP_M, CSP# has a semantic model for shared variables so avoids the need for memory processes that simulate variables. Furthermore, contrasting with FDR, which expands all the specification states before starting an exhaustive checking on the states, PAT implements on-the-fly and partial order reduction techniques that can avoid the complete expansion of the state space and the exhaustive checking of the states. Hence, the usage of PAT can potentially improve the efficiency of our test generation approach if it is used in place of FDR. Nonetheless, CSP# does not support many features of CSP_M like functional programming, set comprehension and implicit process environment. Thus, the representation of sets and use case inputs become verbose. Moreover, in PAT it is not possible to use the functions we have introduced to specify test purposes and sound test cases, since PAT does not offer a functional sublanguage like FDR does. Despite the particularities between CSP_M and CSP#, both languages enable the expression of CSP processes. The choice of CSP_M is mainly due to the convenience of its syntax, which maps the use case elements in a very straightforward manner. Therefore, PAT and FDR have complementary facilities, and a deeper comparative analysis of their advantages and disadvantages for supporting our approach still needs to be done.

7. Conclusions

This paper introduced an automatic test generation approach that inputs use cases structured as document templates and described with textual sentences that follow a controlled natural language standard. Document templates are extended to allow include and extension relations between use cases and to include data elements as user defined types, variables and parameters. Data elements can be used in the feature descriptions to specify control flow behaviour and describe explicit input and output values. The extension does not interfere with the original natural language standard because data elements are included as annotations. In the same way the document is extended, we show incrementally how the mapping from use cases to CSP_M is improved to cope with the new elements of the document. This is easily achieved by modelling variables as memory processes with read and write operations, which are composed with the processes that represent the use case control flow. The syntactic structure of the memory model enables the extension of CSP test purposes to describe test scenarios that match particular states of the use cases in addition to particular traces. Test scenarios are generated using the FDR refinement checker tool to verify traces refinement expressions. Refinement counter-examples are the test scenarios of interest and the input for the construction of sound test cases. A function, $TC_BUILDER$, is defined to yield a sound test case for a given test scenario. Soundness is addressed according to the **cspio** conformance relation, which defines the class of valid implementations for the I/O process that specifies the features behaviour. Alternatively, for contexts where the implementation model is known, we devised a refinement expression that can be automatically checked by FDR and asserts whether an implementation conforms to a given specification.

As already discussed, there are several related approaches to generate test cases from use cases, to represent variables in CSP and to generate test cases considering particular traces and scenarios. However, to our knowledge, a distinguishing feature of our approach is generating test cases from the combination of a natural language representation that combines control and state representation, selection of particular scenarios during test generation and formal properties of the generated tests, like soundness, apart from an automated and sound strategy to check conformance based on process refinement in the CSP traces model.

Adopting the proposed template, simple use cases can be combined to construct more elaborate patterns of behaviour using constructors that relate use case steps (from and to steps) as well as complete use cases (inclusion and extension relations). Such constructors allow very succinct, accurate and high level descriptions of the system to be tested, refraining the specifier from knowing the underlying CSP formal model.

Although the design of the proposed use case templates has been motivated by the mobile devices domain, they can be used to describe other domains of applications, which can be modelled in terms of features (for instance, desktop applications). The same applies to the test generation approach that considers particular states of the model (Sect. 4.9), provided the model represents variables as processes and uses channels to read and update the variables. Apart from the selection of particular states, the generation and selection approaches (Sect. 3) do apply to any input–output CSP model that obeys the relevant alphabet restrictions.

An important aspect to emphasise is the extensibility of our approach, particularly when migrating from a test generation strategy that addressed only control to the one considering state. While the templates and the corresponding CSP models were substantially extended to capture data, surprisingly, perhaps, the test generation strategy, based on counter-examples of refinement checking, was entirely reused. This contrasts with approaches based on more operational models [JJ05, CJRZ01] where an explicit algorithm is developed to address each facet.

Another distinguishing feature of our approach, as already mentioned, is a strategy for automatic conformance verification, as captured by Theorem 5.1. We have further investigated this issue and proved some compositionality results for conformance verification. Particularly, we found out that, if the specification is input complete, **cspio** is a compositional relation for all CSP operators.

The notation of CSP_M was used as the syntax for defining data in the use case document, although it is not very user friendly. One ongoing work [Bez11] is to define a natural language syntax suitable for the specification and manipulation of use case data. Such a syntax combined with that of [Tor06] enables one to define state based use cases entirely in natural language. A similar syntax can be defined to express CSP test purposes in natural language, making the usage of formal methods totally transparent to the user.

So far we have explored only sequential features that do not share data. An immediate improvement is to consider concurrent features with data sharing. In principle, the adaptation of the templates and the translation into CSP to support parallelism will impact only the CSP model of variable access and the composition of features. Since features, in our current model, would allow variables to be composed in parallel, memory read and write operations will consider that variables can be accessed in any order which can lead to race conditions. Therefore, some mutual exclusion mechanism must be included, and flow and memory models would be adapted to control the access to the memory, in a similar way as reported in [Ros11].

Allowing data sharing among distinct features can be achieved by including a memory process that keeps global values and synchronises with the composition of features (see Sect. 4.8), similarly to the proposed solution for features and use case memories. Likewise, the template constructs for the specification of global data would be the same used to declare data local to features and use cases. Actually, we do not foresee the need of any modification in the test generation or selection strategies to handle these extensions.

Our overall strategy is currently implemented into two separate tools. The automatic translation from use case templates to CSP is implemented as a component of the TaRGeT [FNSB10] framework. Test case generation and selection from the CSP model is implemented in a tool called ATG [NSM11]. A current task is to incorporate ATG into the TaRGeT framework.

Because the FDR tool enumerates data, which easily leads to state space explosion, we plan to apply the abstraction approach in [MBS02, DFM09] to automatically transform infinite CSP models into finite ones with behaviour preservation. Preservation of safety behaviour is enough for our models because our entire approach is based on the CSP traces model. A complementary approach is to specify the state based use cases in CSP# and adapt our approach to be run in the PAT model checker. Potentially, the partial order reduction and on-the-fly techniques implemented in the PAT will improve the efficiency of our test generation approach when compared to the FDR tool.

Apart from dealing with control and state, we also plan to extend our approach with time aspects. In this direction, a tool like PAT seems to be a promising alternative to explore, as it already supports modelling and analysis involving time.

Acknowledgments

We thank Bill Roscoe for discussions concerning the design of an efficient memory model in CSP for analysis using FDR. We also thank the Brazilian research agencies CAPES and CNPq for financial support, and particularly CNPq for the grant that supports the INES project. Finally, we thank the anonymous referees for several comments and corrections that significantly contributed to improve this paper.

A. Proofs

A.1. Proof of Theorem 5.1

The following Lemmas are auxiliary to the proof of Theorem 5.1.

Lemma A.1 (Traces of ANY(.))

$$\mathcal{T}(ANY(evset, next)) = \{\langle\rangle\} \cup \{\langle e\rangle \frown t \mid e \in evset \wedge t \in \mathcal{T}(next)\}$$

Proof

$$\begin{aligned}
& \mathcal{T}(ANY(evset, next)) \\
= \; & [\text{ def. } ANY(.)\ ] \\
& \mathcal{T}(\Box\, ev : evset \bullet ev \rightarrow next) \\
= \; & [\text{ def. } \Box\, x : A \bullet F(x) \text{ and } evset = \{e_1, \ldots, e_k\}] \\
& \mathcal{T}(e_1 \rightarrow next \;\Box \cdots \Box\; e_k \rightarrow next) \\
= \; & [\text{ def. } \mathcal{T}(P \,\Box\, Q)] \\
& \mathcal{T}(e_1 \rightarrow next) \cup \cdots \cup \mathcal{T}(e_k \rightarrow next) \\
= \; & [\text{ def. } \mathcal{T}(a \rightarrow P)] \\
& \{\langle\rangle\} \cup \{\langle e_1\rangle \frown t \mid t \in \mathcal{T}(next)\} \cup \cdots \cup \{\langle\rangle\} \cup \{\langle e_k\rangle \frown t \mid t \in \mathcal{T}(next)\} \\
= \; & [\text{ set comprehension}] \\
& \{\langle\rangle\} \cup \{\langle e\rangle \frown t \mid e \in evset \wedge t \in \mathcal{T}(next)\}
\end{aligned}$$

Lemma A.2 (Initials of P interrupted). Let P be a CSP process, and A a set of events. Then

$$initials((P \triangle ANY(A, Stop))/s) = initials(P/s) \cup \{a \in A \mid s \in \mathcal{T}(P) \cap \Sigma^*\}$$

Proof

$$\begin{aligned}
& initials((P \triangle ANY(A, Stop))/s) \\
= \; & [\text{ from Defs. of } \mathcal{T}(P/s) \text{ and } initials(P)\ ] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P \triangle ANY(A, Stop))\} \\
= \; & [\text{ def. } \mathcal{T}(P \triangle Q)\ ] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P) \cup \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge w \in \mathcal{T}(ANY(A, Stop))\}\} \\
= \; & [\text{ set comprehension }] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge w \in \mathcal{T}(ANY(A, Stop))\}\} \\
= \; & [\text{ Lemma A.1 and } \mathcal{T}(Stop)\ ] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge w \in \{\langle\rangle\} \cup \{\langle a\rangle \mid a \in A\}\}\} \\
= \; & [\text{ set comprehension }] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{\langle\rangle \frown w \mid w \in \{\langle a\rangle \mid a \in A\}\} \;\cup \\
& \qquad \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge w = \langle\rangle\} \;\cup \\
& \qquad \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge w \in \{\langle a\rangle \mid a \in A\}\}\} \\
= \; & [\text{ set comprehension }] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{\langle\rangle \frown w \mid w \in \{\langle a\rangle \mid a \in A\}\}\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge w = \langle\rangle\}\} \;\cup \\
& \{e \mid s \frown \langle e\rangle \in \{t \frown w \mid t \in \mathcal{T}(P) \cap \Sigma^* \wedge t \neq \langle\rangle \wedge w \in \{\langle a\rangle \mid a \in A\}\}\}
\end{aligned}$$

$$
\begin{aligned}
= \;& [\text{ set comprehension }] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \cup \\
& \{e \mid s = \langle\rangle \wedge e \in A\} \cup \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P) \cap \Sigma^*\} \cup \\
& \{e \mid s \in \mathcal{T}(P) \cap \Sigma^* \wedge e \in A\} \\
= \;& [\; A \subseteq B \equiv A \cup B = B \text{ and } \mathcal{T}(P) \cap \Sigma^* \subseteq \mathcal{T}(P) \;] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \cup \\
& \{e \mid s = \langle\rangle \wedge e \in A\} \cup \\
& \{e \mid s \in \mathcal{T}(P) \cap \Sigma^* \wedge e \in A\} \\
= \;& [\; A \subseteq B \equiv A \cup B = B \text{ and } \langle\rangle \in \mathcal{T}(P) \cap \Sigma^* \;] \\
& \{e \mid s \frown \langle e\rangle \in \mathcal{T}(P)\} \cup \\
& \{e \mid s \in \mathcal{T}(P) \cap \Sigma^* \wedge e \in A\} \\
= \;& [\text{ def. of } \mathit{initials}(P/s) \;] \\
& \mathit{initials}(P/s) \cup \{e \mid s \in \mathcal{T}(P) \cap \Sigma^* \wedge e \in A\} \\
= \;& [\text{ set comprehension }] \\
& \mathit{initials}(P/s) \cup \{a \in A \mid s \in \mathcal{T}(P) \cap \Sigma^*\}
\end{aligned}
$$

The proof of Theorem 5.1.

Theorem 5.1 (Verification of cspio). Let $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ be an implementation model, and $S = (P_S, A_{I_S}, A_{O_S})$ a specification, with $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. Then IUT **cspio** S holds iff the following refinement holds.

$$P_S \sqsubseteq_\tau (P_S \triangle ANY(A_{O_{IUT}}, Stop)) \;|[A_{I_{IUT}} \cup A_{O_{IUT}}]|\; P_{IUT}$$

Proof

$$
\begin{aligned}
& P_S \sqsubseteq_\tau (P_S \triangle ANY(A_{O_{IUT}}, Stop)) \,|[A_{I_{IUT}} \cup A_{O_{IUT}}]|\, P_{IUT} \\
= \;& [\text{ definition of } \sqsubseteq_\tau \;] \\
& \mathcal{T}((P_S \triangle ANY(A_{O_{IUT}}, Stop)) \,|[A_{I_{IUT}} \cup A_{O_{IUT}}]|\, P_{IUT}) \subseteq \mathcal{T}(P_S) \\
= \;& [\; A_{I_S} \subseteq A_{I_{IUT}}, A_{O_S} \subseteq A_{O_{IUT}}, \mathcal{T}(P \,|[\alpha_P \cup \alpha_Q]|\, Q) = \mathcal{T}(P \parallel Q) \equiv \alpha_P \subseteq \alpha_Q \\
& \;\text{ and } \mathcal{T}(P \parallel Q) = \mathcal{T}(P) \cap \mathcal{T}(Q) \;] \\
& \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \subseteq T(P_S) \\
= \;& [\text{ definition } \subseteq \;] \\
& \forall\, s \bullet s \in \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \Rightarrow s \in \mathcal{T}(P_S) \\
= \;& [\text{ holds when sequence is empty or neither }] \\
& \langle\rangle \in \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \Rightarrow \langle\rangle \in \mathcal{T}(P_S) \wedge \\
& \forall\, s, x \bullet s \frown \langle x\rangle \in \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \Rightarrow \\
& \quad s \frown \langle x\rangle \in \mathcal{T}(P_S) \\
= \;& [\text{ traces property } \forall\, P \bullet \langle\rangle \in \mathcal{T}(P) \;] \\
& \mathit{true} \Rightarrow \mathit{true} \wedge \\
& \forall\, s, x \bullet s \frown \langle x\rangle \in \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \Rightarrow \\
& \quad s \frown \langle x\rangle \in \mathcal{T}(P_S) \\
= \;& [\; \wedge \text{ elimination }] \\
& \forall\, s, x \bullet s \frown \langle x\rangle \in \mathcal{T}(P_S \triangle ANY(A_{O_{IUT}}, Stop)) \cap \mathcal{T}(P_{IUT}) \Rightarrow \\
& \quad s \frown \langle x\rangle \in \mathcal{T}(P_S) \\
= \;& [\text{ def. } \mathit{initials}(.) \text{ and } \mathcal{T}(P/s) \;] \\
& \forall\, s, x \bullet x \in \mathit{initials}((P_S \triangle ANY(A_{O_{IUT}}, Stop))/s) \cap \mathit{initials}(P_{IUT}/s) \Rightarrow \\
& \qquad x \in \mathit{initials}(P_S/s)
\end{aligned}
$$

$$
\begin{aligned}
= \;& [\text{ Lemma A.2 }]\\
& \forall\, s, x \bullet x \in (\mathit{initials}(P_S/s) \cup \{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\}) \cap\\
& \qquad \mathit{initials}(P_{IUT}/s) \Rightarrow x \in \mathit{initials}(P_S/s)\\
= \;& [\, \cap\text{-dist-}\cup \,]\\
& \forall\, s, x \bullet x \in (\mathit{initials}(P_S/s) \cap \mathit{initials}(P_{IUT}/s)) \cup\\
& \qquad (\{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\} \cap \mathit{initials}(P_{IUT}/s)) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\text{ def } \cup \,]\\
& \forall\, s, x \bullet x \in \mathit{initials}(P_S/s) \cap \mathit{initials}(P_{IUT}/s) \vee\\
& \qquad x \in \{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\} \cap \mathit{initials}(P_{IUT}/s) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\, A \vee B \Rightarrow C \equiv A \Rightarrow C \wedge B \Rightarrow C \,]\\
& \forall\, s, x \bullet x \in \mathit{initials}(P_S/s) \cap \mathit{initials}(P_{IUT}/s) \Rightarrow x \in \mathit{initials}(P_S/s) \wedge\\
& \qquad x \in \{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\} \cap \mathit{initials}(P_{IUT}/s) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\, x \in A \cap B \Rightarrow x \in A \equiv A \cap B \subseteq A \equiv \mathit{true} \,]\\
& \forall\, s, x \bullet \mathit{true} \wedge\\
& \qquad x \in \{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\} \cap \mathit{initials}(P_{IUT}/s) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\, \wedge \text{ elimination }]\\
& \forall\, s, x \bullet x \in \{a \in A_{O_{IUT}} \mid s \in \mathcal{T}(P_S) \cap \Sigma^*\} \cap \mathit{initials}(P_{IUT}/s) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\text{ predicate logics }]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^*;\; x \bullet x \in A_{O_{IUT}} \cap \mathit{initials}(P_{IUT}/s) \Rightarrow\\
& \qquad x \in \mathit{initials}(P_S/s)\\
= \;& [\text{ def } \mathit{out}(.) \,]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^*;\; x \bullet x \in \mathit{out}(IUT, s) \Rightarrow x \in \mathit{initials}(P_S/s)\\
= \;& [\text{ set theory }]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^* \bullet \mathit{out}(IUT, s) \subseteq \mathit{initials}(P_S/s)\\
= \;& [\, \Sigma = A_{I_{IUT}} \cup A_{O_{IUT}} \text{ and } \mathit{initials}(P_S) \subseteq \Sigma \,]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^* \bullet \mathit{out}(IUT, s) \subseteq \mathit{initials}(P_S/s) \cap (A_{I_{IUT}} \cup A_{O_{IUT}})\\
= \;& [\, \cap\text{-dist-}\cup \,]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^* \bullet \mathit{out}(IUT, s) \subseteq (\mathit{initials}(P_S/s) \cap A_{I_{IUT}}) \cup\\
& \qquad (\mathit{initials}(P_S/s) \cap A_{O_{IUT}})\\
= \;& [\text{ def. } \mathit{out}(.) \,]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^* \bullet \mathit{out}(IUT, s) \subseteq (\mathit{initials}(P_S/s) \cap A_{I_{IUT}}) \cup \mathit{out}(S, s)\\
= \;& [\, A_{I_{IUT}} \cap A_{O_{IUT}} = \emptyset \text{ and set theory }]\\
& \forall\, s : \mathcal{T}(P_S) \cap \Sigma^* \bullet \mathit{out}(IUT, s) \subseteq \mathit{out}(S, s)\\
= \;& [\text{ because } \checkmark \notin \mathit{out}(M, s) \,]\\
& \forall\, s : \mathcal{T}(P_S) \bullet \mathit{out}(IUT, s) \subseteq \mathit{out}(S, s)\\
= \;& [\text{ Definition 5 }]\\
& IUT \;\textbf{cspio}\; S
\end{aligned}
$$

◇

A.2. Proof of Theorem 5.2

The following Lemmas are auxiliary to the proof of Theorem 5.2.

Lemma A.3 (Traces of SUBTC(.)). Let $S = (P_S, A_{I_S}, A_{O_S})$ be a specification I/O process. The traces of the auxiliary process $SUBTC((e, outs))$, defined in Sect. 5, is

$$\mathcal{T}(SUBTC((e, outs)) = \{\langle\rangle, \langle e\rangle, \langle e, \checkmark\rangle\} \cup \{ev \frown t \mid ev \in outs - \{e\} \wedge t \in \{\langle\rangle, \langle inc\rangle\} \wedge e \in A_{O_S}\}$$
$$\cup \{ev \frown t \mid ev \in A_{O_I UT} - outs \wedge t \in \{\langle\rangle, \langle fail\rangle\} \wedge e \in A_{O_S}\}$$

Proof

$$\begin{array}{ll}
 & \mathcal{T}(SUBTC((e, outs))) \\
= & [\text{ def. } SUBTC(.) \text{ in Sect. 5 }] \\
 & \mathcal{T}(\ e \rightarrow\ Skip \,\Box\, e \in A_{O_S} \,\&\, (ANY(outs - \{e\},\ INC\ ) \,\Box\, ANY(A_{O_{IUT}} - outs,\ FAIL\ ))) \\
= & [\text{ defs. } \mathcal{T}(P \,\Box\, Q)\ ] \\
 & \mathcal{T}(e \rightarrow Skip) \cup \mathcal{T}(e \in A_{O_S} \,\&\, (ANY(outs - \{e\},\ INC\ ) \,\Box\, ANY(A_{O_{IUT}} - outs,\ FAIL\ ))) \\
= & [\text{ def. } b \,\&\, P\ ] \\
 & \mathcal{T}(e \rightarrow Skip) \cup \\
 & \mathcal{T}(if (e \in A_{O_S})\ then\ ANY(outs - \{e\},\ INC\ ) \,\Box\, ANY(A_{O_{IUT}} - outs,\ FAIL\ )\ else\ Stop) \\
= & [\text{ def. of } \mathcal{T}(if\ (b)\ then\ P\ else\ Q) \text{ and } \mathcal{T}(Stop) \cup \mathcal{T}(P) = \mathcal{T}(P)\ ] \\
 & \mathcal{T}(e \rightarrow Skip) \cup \\
 & \{t \mid t \in \mathcal{T}(ANY(outs - \{e\},\ INC\ ) \,\Box\, ANY(A_{O_{IUT}} - outs,\ FAIL\ )) \wedge e \in A_{O_S}\} \\
= & [\text{ def. } \mathcal{T}(P \,\Box\, Q)\ ] \\
 & \mathcal{T}(e \rightarrow Skip) \cup \\
 & \{t \mid t \in \mathcal{T}(ANY(outs - \{e\},\ INC\ )) \cup \mathcal{T}(ANY(A_{O_{IUT}} - outs,\ FAIL\ )) \wedge e \in A_{O_S}\} \\
= & [\text{ set comprehension }] \\
 & \mathcal{T}(e \rightarrow Skip) \cup \\
 & \{t \mid t \in \mathcal{T}(ANY(outs - \{e\},\ INC\ )) \wedge e \in A_{O_S}\} \cup \\
 & \{t \mid t \in \mathcal{T}(ANY(A_{O_{IUT}} - outs,\ FAIL\ )) \wedge e \in A_{O_S}\} \\
= & [\text{ Lemma A.1, def. } \mathcal{T}(e \rightarrow Skip), \text{ def. } \mathcal{T}(INC) \text{ and def. } \mathcal{T}(FAIL)\ ] \\
 & \{\langle\rangle, \langle e\rangle, \langle e, \checkmark\rangle\} \cup \\
 & \{\langle ev\rangle \frown t \mid ev \in outs - \{e\} \wedge t \in \{\langle\rangle, \langle inc\rangle\} \wedge e \in A_{O_S}\} \cup \\
 & \{\langle ev\rangle \frown t \mid ev \in A_{O_{IUT}} - outs \wedge t \in \{\langle\rangle, \langle fail\rangle\} \wedge e \in A_{O_S}\}
\end{array}$$

Lemma A.4 (Traces of TC_BUILDER(.)). Let $S = (P_S, A_{I_S}, A_{O_S})$ be a specification I/O process, $ts = \langle e_1, \ldots, e_{\#}\rangle$ a test scenario from S, such that $\#ts > 0$. Moreover, $atrace = \langle (e_1, outs_1^{ts}), \ldots, (e_{\#ts}, outs_{\#ts}^{ts})\rangle$ is an annotated trace obtained from ts then $\mathcal{T}(TC_BUILDER(atrace))$ is

$$\{\langle\rangle\} \cup \{\langle ev\rangle \frown t \mid ev \in outs_1 - \{e_1\} \wedge t \in \{\langle\rangle, \langle inc\rangle\} \wedge e_1 \in A_{O_S}\} \cup$$
$$\{\langle ev\rangle \frown t \mid ev \in A_{O_{IUT}} - outs_1 \wedge t \in \{\langle\rangle, \langle fail\rangle\} \wedge e_1 \in A_{O_S}\} \cup$$
$$\{\langle e_1\rangle \frown t \mid t \in \mathcal{T}(TC_BUILDER(tail(atrace)))\}$$

Proof

$$\begin{array}{ll}
 & \mathcal{T}(TC_BUILDER(atrace)) \\
= & [\text{ def. } TC_BUILDER(.) \text{ in Sect. 5 }] \\
 & \mathcal{T}(SUBTC((e_1, outs_1));\ TC_BUILDER(tail(atrace))) \\
= & [\text{ def. } \mathcal{T}(P;\ Q)\ ] \\
 & \mathcal{T}(SUBTC((e_1, outs_1))) \cap \Sigma^* \cup \\
 & \{s \frown t \mid s \frown \langle\checkmark\rangle \in \mathcal{T}(SUBTC((e_1, outs_1))) \wedge t \in \mathcal{T}(TC_BUILDER(tail(atrace)))\} \\
= & [\text{ according to Lemma A.3 we have } s \frown \langle\checkmark\rangle \in \mathcal{T}(SUBTC((e_1, outs_1))) \equiv s = \langle e_1\rangle\ ] \\
 & \mathcal{T}(SUBTC((e_1, outs_1))) \cap \Sigma^* \cup \\
 & \{\langle e_1\rangle \frown t \mid t \in \mathcal{T}(TC_BUILDER(tail(atrace)))\}
\end{array}$$

$$= \quad [\text{ Lemma A.3 and } \{t \frown \langle\checkmark\rangle\} \cap \Sigma^* = \{t\}]$$

$$\{\langle\rangle\} \cup \{\langle ev\rangle \frown t \mid ev \in outs_1 - \{e_1\} \wedge t \in \{\langle\rangle, \langle inc\rangle\} \wedge e_1 \in A_{O_S}\} \cup$$

$$\{\langle ev\rangle \frown t \mid ev \in A_{O_{IUT}} - outs_1 \wedge t \in \{\langle\rangle, \langle fail\rangle\} \wedge e_1 \in A_{O_S}\} \cup$$

$$\{\langle e_1\rangle \frown t \mid t \in \mathcal{T}(TC_BUILDER(tail(atrace)))\}$$

Recalling from Sect. 5, $outs_i$ is the set of output events offered by the specification process component P_S after the prefix $\langle ev_1, \ldots, ev_{i-1}\rangle$ of a test scenario $ts = \langle ev_1, \ldots, ev_{\#ts}\rangle$, $out(S, \langle ev_1, \ldots, ev_{i-1}\rangle)$. Moreover, consider that $sufixes(s)$ is the function that yields the suffixes of sequence s defined as

$$sufixes(\langle\rangle) = \{\langle\rangle\}$$

$$sufixes(\langle e\rangle \frown t) = \{\langle e\rangle \frown t\} \cup sufixes(t)$$

In Lemma A.5, we use the notation $outs_i^t$ to denote the outputs produced by the specification process component after a suffix $t = \langle e_1, \ldots, e_{i-1}\rangle$ of ts ($t \in sufixes(ts)$), formally,

$$\begin{aligned} outs_i^t = {} & if((\exists\, s \bullet s \frown t = ts) \wedge t \neq \langle\rangle)\ then \\ & \quad if(t = ts)\ then\ outs_i \\ & \quad else\ out(S, s \frown \langle e_1, \ldots, e_{i-1}\rangle) \\ & else\ \emptyset \end{aligned}$$

This function is an extension of $outs_i$ to consider suffixes of a test scenario ts. It is equivalent to $outs_i$ if the suffix equals to the test scenario.

Lemma A.5 (Failure traces of a test case). Let $S = (P_S, A_{I_S}, A_{O_S})$ be a specification I/O process and let $AS(ts)$ be a function that yields $atrace = \langle(e_1, outs_1), \ldots, (e_{\#ts}, outs_{\#ts})\rangle$, the annotated sequence obtained from the test scenario $ts = \langle e_1, \ldots, e_{\#ts}\rangle$ of P_S. Moreover, consider $prefixes(s)$ is the function that yields the prefixes of sequence s defined as

$$prefixes(\langle\rangle) = \{\langle\rangle\}$$

$$prefixes(t \frown \langle e\rangle) = prefixes(t) \cup \{t \frown \langle e\rangle\}$$

Then

$$\forall\, t : \mathcal{T}\ (TC_BUILDER(AS(ts))) \mid t \restriction VER = \langle fail\rangle \bullet$$
$$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(ts) - \{ts\} \wedge o \in A_{O_{IUT}} - outs^{ts}_{\#s+1}$$

Proof

The proof for $ts = \langle\rangle$ is trivial.

$$\equiv \quad [\,\forall\, t \mid P(t) \bullet Q(t) \equiv \forall\, t \bullet P(t) \Rightarrow Q(t)\,]$$
$$t \in \mathcal{T}(TC_BUILDER(\langle\rangle) \bullet t \restriction VER = \langle fail\rangle \Rightarrow$$
$$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$$

$$\equiv \quad [\ TC_BUILDER(\langle\rangle) = PASS \text{ and } \mathcal{T}(PASS)\ ]$$
$$t \in \{\langle\rangle, \langle pass\rangle\} \bullet t \restriction VER = \langle fail\rangle \Rightarrow$$
$$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$$

$$\equiv \quad [\text{ predicate logics }]$$
$$\langle\rangle \restriction VER = \langle fail\rangle \Rightarrow$$
$$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$$
$$\wedge$$
$$\langle pass\rangle \restriction VER = \langle fail\rangle \Rightarrow$$
$$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$$

$\equiv$ [def. of $s \upharpoonright X$]

$false \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$

$\wedge$

$false \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle\rangle) - \{\langle\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle\rangle}_{\#s+1}$

$\equiv$ [predicate logics]

$true$

For $\#ts > 0$ we prove using induction. The base case is $ts = \langle e_1\rangle$.

$\equiv$ [$\forall t \mid P(t) \bullet Q(t) \equiv \forall t \bullet P(t) \Rightarrow Q(t)$]

$t \in \mathcal{T}(TC_BUILDER(AS(\langle e_1\rangle))) \bullet$

$t \upharpoonright VER = \langle fail\rangle \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs^{\langle e_1\rangle}_{\#s+1}$

$\equiv$ [$ts = \langle e_1\rangle \equiv outs^{\langle e_1\rangle}_i = outs_i$]

$t \in \mathcal{T}(TC_BUILDER(AS(\langle e_1\rangle))) \bullet$

$t \upharpoonright VER = \langle fail\rangle \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\equiv$ [def. $AS(.)$ and Lemma A.4]

$t \in \{\langle\rangle\} \cup \{\langle ev\rangle \frown t' \mid ev \in outs_1 - \{e_1\} \wedge t' \in \{\langle\rangle, \langle inc\rangle\} \wedge e_1 \in A_{O_S}\} \cup$

$\{\langle ev\rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail\rangle\} \wedge e_1 \in A_{O_S}\} \cup$

$\{\langle e_1\rangle \frown t' \mid t' \in \mathcal{T}(TC_BUILDER(\langle\rangle))\} \bullet$

$t \upharpoonright VER = \langle fail\rangle \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\equiv$ [defs. $TC_BUILDER(\langle\rangle)$ and $\mathcal{T}(PASS)$]

$t \in \{\langle\rangle\} \cup \{\langle ev\rangle \frown t' \mid ev \in outs_1 - \{e_1\} \wedge t' \in \{\langle\rangle, \langle inc\rangle\} \wedge e_1 \in A_{O_S}\} \cup$

$\{\langle ev\rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail\rangle\} \wedge e_1 \in A_{O_S}\} \cup$

$\{\langle e_1\rangle \frown t' \mid t' \in \{\langle\rangle, \langle pass\rangle\}\} \bullet$

$t \upharpoonright VER = \langle fail\rangle \Rightarrow t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\equiv$ [$(x \in A \cup B \bullet P(x) \Rightarrow Q(x)) \equiv ((x \in A \wedge P(x)) \Rightarrow Q(x)) \wedge ((x \in B \wedge P(x)) \Rightarrow Q(x))$]

$t \in \{\langle\rangle\} \wedge t \upharpoonright VER = \langle fail\rangle \Rightarrow$

$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\wedge$

$t \in \{\langle ev\rangle \frown t' \mid ev \in outs_1 - \{e_1\} \wedge t' \in \{\langle\rangle, \langle inc\rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail\rangle \Rightarrow$

$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\wedge$

$t \in \{\langle ev\rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail\rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail\rangle \Rightarrow$

$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$\wedge$

$t \in \{\langle e_1\rangle \frown t' \mid t' \in \{\langle\rangle, \langle pass\rangle\}\} \wedge t \upharpoonright VER = \langle fail\rangle \Rightarrow$

$t = s \frown \langle o, fail\rangle \wedge s \in prefixes(\langle e_1\rangle) - \{\langle e_1\rangle\} \wedge o \in A_{O_{IUT}} - outs_{\#s+1}$

$$
\begin{aligned}
\equiv\quad & [\text{ set comprehension and def. } s \upharpoonright X \text{]}\\
& \mathit{false} \Rightarrow t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
& \wedge\\
& \mathit{false} \Rightarrow t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
& \wedge\\
& t \in \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge t' \in \{\langle\rangle, \langle \mathit{fail} \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \Rightarrow\\
& \quad t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
& \wedge\\
& \mathit{false} \Rightarrow t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
\equiv\quad & [\text{ propositional logics]}\\
& t \in \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge t' \in \{\langle\rangle, \langle \mathit{fail} \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \Rightarrow\\
& \quad t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
\equiv\quad & [\text{ set comprehension]}\\
& t = \langle ev \rangle \vee t = \langle ev, \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \Rightarrow\\
& \quad t = s \frown \langle o, \mathit{fail} \rangle \wedge s \in \mathit{prefixes}(\langle e_1 \rangle) - \{\langle e_1 \rangle\} \wedge o \in A_{O_{IUT}} - \mathit{outs}_{\#s+1}\\
\equiv\quad & [\ s \in \mathit{prefixes}(\langle e \rangle) - \{\langle e \rangle\} \equiv s = \langle\rangle\ ]\\
& t = \langle ev \rangle \vee t = \langle ev, \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \Rightarrow\\
& \quad t = \langle o, \mathit{fail} \rangle \wedge o \in A_{O_{IUT}} - \mathit{outs}_1\\
\equiv\quad & [\ \wedge - \mathit{dist} - \vee\ ]\\
& (t = \langle ev \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle) \vee\\
& (t = \langle ev, \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle) \Rightarrow\\
& \quad t = \langle o, \mathit{fail} \rangle \wedge o \in A_{O_{IUT}} - \mathit{outs}_1\\
\equiv\quad & [\ \langle ev \rangle \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} \equiv \mathit{false}\ ]\\
& \mathit{false} \vee\\
& (t = \langle ev, \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle) \Rightarrow\\
& \quad t = \langle o, \mathit{fail} \rangle \wedge o \in A_{O_{IUT}} - \mathit{outs}_1\\
\equiv\quad & [\text{ propositional logics]}\\
& t = \langle ev, \mathit{fail} \rangle \wedge ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle \Rightarrow\\
& \quad t = \langle o, \mathit{fail} \rangle \wedge o \in A_{O_{IUT}} - \mathit{outs}_1\\
\equiv\quad & [\ P \wedge Q \Rightarrow P\ ]\\
& \mathit{true}
\end{aligned}
$$

For the inductive case ($ts = \langle e_1 \rangle \frown ts'$), we have.

$$
\begin{aligned}
& t \in \mathcal{T}(\mathit{TC_BUILDER}(\mathit{AS}(\langle e_1 \rangle \frown ts'))) \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle\\
\equiv\quad & [\text{ def. } \mathit{AS}(.) \text{ and Lemma A.4]}\\
& t \in \{\langle\rangle\} \cup \{\langle ev \rangle \frown t' \mid ev \in \mathit{outs}_1 - \{e_1\} \wedge t' \in \{\langle\rangle, \langle \mathit{inc} \rangle\} \wedge e_1 \in A_{O_S}\} \cup\\
& \qquad \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - \mathit{outs}_1 \wedge t' \in \{\langle\rangle, \langle \mathit{fail} \rangle\} \wedge e_1 \in A_{O_S}\} \cup\\
& \qquad \{\langle e_1 \rangle \frown t' \mid t' \in \mathcal{T}(\mathit{TC_BUILDER}(\mathit{AS}(ts')))\} \wedge t \upharpoonright \mathit{VER} = \langle \mathit{fail} \rangle
\end{aligned}
$$

$$
\begin{array}{ll}
\equiv & [\ (x \in A \cup B \wedge P(x)) \equiv (x \in A \wedge P(x)) \vee (x \in B \wedge P(x))\ ] \\
& t \in \{\langle\rangle\} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t \in \{\langle ev \rangle \frown t' \mid ev \in outs_1 - \{e_1\} \wedge t' \in \{\langle\rangle, \langle inc \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t \in \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t \in \{\langle e_1 \rangle \frown t' \mid t' \in \mathcal{T}(TC_BUILDER(AS(ts')))\} \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ (\langle\rangle \upharpoonright VER = \langle fail \rangle) \equiv (\langle ev \rangle \upharpoonright VER = \langle fail \rangle \wedge ev \in A_{O_{IUT}}) \equiv \\
& (\langle ev, inc \rangle \upharpoonright VER = \langle fail \rangle \wedge ev \in A_{O_{IUT}}) \equiv false\ ] \\
& false \\
& \vee \\
& false \\
& \vee \\
& t \in \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t \in \{\langle e_1 \rangle \frown t' \mid t' \in \mathcal{T}(TC_BUILDER(AS(ts')))\} \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ \text{propositional logics}\ ] \\
& t \in \{\langle ev \rangle \frown t' \mid ev \in A_{O_{IUT}} - outs_1 \wedge t' \in \{\langle\rangle, \langle fail \rangle\} \wedge e_1 \in A_{O_S}\} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t \in \{\langle e_1 \rangle \frown t' \mid t' \in \mathcal{T}(TC_BUILDER(AS(ts')))\} \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ \text{set comprehension}\ ] \\
& t = \langle ev \rangle \vee t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ \wedge -dist- \vee\ ] \\
& (t = \langle ev \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright VER = \langle fail \rangle) \vee \\
& (t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright VER = \langle fail \rangle) \\
& \vee \\
& t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ \langle ev \rangle \upharpoonright VER = \langle fail \rangle \wedge ev \in A_{O_{IUT}} \equiv false\ ] \\
& false \vee \\
& (t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright VER = \langle fail \rangle) \\
& \vee \\
& t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t \upharpoonright VER = \langle fail \rangle \\
\equiv & [\ \text{propositional logics}\ ] \\
& t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \wedge e_1 \in A_{O_S} \wedge t \upharpoonright VER = \langle fail \rangle \\
& \vee \\
& t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t \upharpoonright VER = \langle fail \rangle \\
\Rightarrow & [\ p \wedge q \Rightarrow p\ ] \\
& t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1 \\
& \vee \\
& t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t \upharpoonright VER = \langle fail \rangle
\end{array}
$$

$$\Rightarrow \quad [\, e_1 \in (A_{I_{IUT}} \cup A_{O_{IUT}}) \wedge VER \cap (A_{I_{IUT}} \cup A_{O_{IUT}}) = \emptyset \wedge \langle e_1 \rangle \frown t' \upharpoonright VER = \langle fail \rangle \Rightarrow t' \upharpoonright VER = \langle fail \rangle \,]$$

$$t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1$$
$$\vee$$
$$t = \langle e_1 \rangle \frown t' \wedge t' \in \mathcal{T}(TC_BUILDER(AS(ts'))) \wedge t' \upharpoonright VER = \langle fail \rangle$$

$$\Rightarrow \quad [\text{ induction hypothesis }]$$
$$t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1$$
$$\vee$$
$$t = \langle e_1 \rangle \frown t' \wedge t' = s \frown \langle o, fail \rangle \wedge s \in prefixes(ts') - \{ts'\} \wedge o \in A_{O_{IUT}} - outs^{ts'}_{\#s+1}$$

$$\equiv \quad [\text{ sequence comprehension and def. of } outs^{ts}_i \,]$$
$$t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1$$
$$\vee$$
$$t = \langle e_1 \rangle \frown s \frown \langle o, fail \rangle \wedge s \in prefixes(ts') - \{ts'\} \wedge o \in A_{O_{IUT}} - outs^{\langle e_1 \rangle \frown ts'}_{\#s+2}$$

$$\equiv \quad [\text{ sequence comprehension, def. } outs^{ts}_i \text{ and def. } prefixes(.) \,]$$
$$t = \langle ev, fail \rangle \wedge ev \in A_{O_{IUT}} - outs_1$$
$$\vee$$
$$t = s \frown \langle o, fail \rangle \wedge s \in prefixes(\langle e_1 \rangle \frown ts') - (\{\langle e_1 \rangle \frown ts'\} \cup \{\langle\rangle\}) \wedge o \in A_{O_{IUT}} - outs^{\langle e_1 \rangle \frown ts'}_{\#s+1}$$

$$\equiv \quad [\text{ sequence comprehension and def. } prefixes(.) \,]$$
$$t = s \frown \langle o, fail \rangle \wedge s \in prefixes(\langle e_1 \rangle \frown ts') - \{\langle e_1 \rangle \frown ts'\} \wedge o \in A_{O_{IUT}} - outs^{\langle e_1 \rangle \frown ts'}_{\#s+1}$$

The proof of the Theorem 5.2.

Theorem 5.2 (TC_BUILDER is sound). Let $S = (P_S, A_{I_S}, A_{O_S})$ be a specification, ts a test scenario from S and $IUT = (P_{IUT}, A_{I_{IUT}}, A_{O_{IUT}})$ an implementation model, such that $A_{I_S} \subseteq A_{I_{IUT}}$ and $A_{O_S} \subseteq A_{O_{IUT}}$. If $atrace$ is an annotated trace obtained from ts, then $TC = (TC_BUILDER(atrace), A_{I_{TC}}, A_{O_{TC}})$ is a sound test case, such that $A_{I_{TC}} = A_{O_{IUT}}$ and $A_{O_{TC}} = A_{I_S}$.

Proof

$$\langle fail \rangle \in \mathcal{T}(EX(IUT, TC) \setminus A_{I_{IUT}} \cup A_{O_{IUT}})$$

$$= \quad [\text{ definition } EX(.) \,]$$
$$\langle fail \rangle \in \mathcal{T}(\, (P_{IUT} \parallel [A_{I_{IUT}} \cup A_{O_{IUT}}]|\ TC_BUILDER(atrace)\,) \setminus A_{I_{IUT}} \cup A_{O_{IUT}})$$

$$= \quad [\text{ definition } P\ |[X][Y]|\ Q \text{ in [Ros98], page 68 }]$$
$$\langle fail \rangle \in \mathcal{T}(\, (P_{IUT}\ |[A_{I_{IUT}} \cup A_{O_{IUT}} \mid \alpha_{TC_BUILDER(atrace)}]|\ TC_BUILDER(atrace)\,) \setminus \alpha_{IUT_{CSP}})$$

$$= \quad [\, \alpha_{TC_BUILDER} \subseteq A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER \,]$$
$$\langle fail \rangle \in \mathcal{T}(\, (P_{IUT}\ |[A_{I_{IUT}} \cup A_{O_{IUT}} \mid A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER]|\ TC_BUILDER(atrace)\,)$$

$$= \quad [\text{ definition } P \setminus X \text{ in [Ros98], page 84 }]$$
$$\langle fail \rangle \in \mathcal{T}(\, (P_{IUT}\ |[\alpha_{IUT_{CSP}} \mid \alpha_{IUT_{CSP}} \cup VER]|\ TC_BUILDER(atrace)\,) \upharpoonright (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER) - A_{I_{IUT}} \cup A_{O_{IUT}})$$

$$= \quad [\text{ set theory }]$$
$$\langle fail \rangle \in \mathcal{T}(\, (P_{IUT}\ |[A_{I_{IUT}} \cup A_{O_{IUT}} \mid A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER]|\ TC_BUILDER(atrace)\,)) \upharpoonright VER$$

$$= \quad [\text{ definition } \mathcal{T}(P\ |[X \mid Y]|\ Q) \text{ in [Ros98], page 60}]$$
$$\langle fail \rangle \in \{s \in (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER)^* \mid (s \upharpoonright A_{I_{IUT}} \cup A_{O_{IUT}}) \in \mathcal{T}(P_{IUT}) \wedge (s \upharpoonright (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER)) \in \mathcal{T}(TC_BUILDER(atrace))\} \upharpoonright VER$$

$$= \quad [\text{ definition } \in \,]$$
$$\exists\, s \mid s \in (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER)^* \wedge s \upharpoonright A_{I_{IUT}} \cup A_{O_{IUT}} \in \mathcal{T}(P_{IUT}) \wedge s \upharpoonright (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER) \in \mathcal{T}(TC_BUILDER(atrace)) \bullet s \upharpoonright VER = \langle fail \rangle$$

$\Rightarrow$ [Lemma A.5 and
$(\exists\, x \mid P(x) \wedge Q(x) \bullet R(x)) \wedge (\forall\, x \mid Q(x) \wedge R(x) \bullet S(x)) \Rightarrow \exists\, x \mid P(x) \wedge S(x)$]
$\exists\, s \mid s \in (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER)^* \wedge s \restriction A_{I_{IUT}} \cup A_{O_{IUT}} \in \mathcal{T}(P_{IUT}) \wedge$
$s = s' \frown \langle o, fail \rangle \wedge s' \in prefixes(ts) - \{ts\} \wedge o \in A_{O_{IUT}} - outs^{ts}_{\#s'+1}$

$\equiv$ [$outs^{ts}_i = outs_i$]
$\exists\, s \mid s \in (A_{I_{IUT}} \cup A_{O_{IUT}} \cup VER)^* \wedge s \restriction A_{I_{IUT}} \cup A_{O_{IUT}} \in \mathcal{T}(P_{IUT}) \wedge$
$s = s' \frown \langle o, fail \rangle \wedge s' \in prefixes(ts) - \{ts\} \wedge o \in A_{O_{IUT}} - outs_{\#s'+1}$

$\Rightarrow$ [$\wedge$-elimination]
$\exists\, s \mid s \restriction A_{I_{IUT}} \cup A_{O_{IUT}} \in \mathcal{T}(P_{IUT}) \wedge s = s' \frown \langle o, fail \rangle \wedge s' \in prefixes(ts) - \{ts\} \wedge$
$o \in A_{O_{IUT}} - outs_{\#s'+1}$

$\Rightarrow$ [def. $s \restriction X$]
$\exists\, s \mid s \in \mathcal{T}(P_{IUT}) \wedge s = s' \frown \langle o \rangle \wedge s' \in prefixes(ts) - \{ts\} \wedge o \in A_{O_{IUT}} - outs_{\#s'+1}$

$\Rightarrow$ [$ts \in \mathcal{T}(P_S) \wedge s' \in prefixes(ts) - \{ts\} \Rightarrow s' \in \mathcal{T}(P_S)$]
$\exists\, s \mid s \in \mathcal{T}(P_{IUT}) \wedge s = s' \frown \langle o \rangle \wedge o \in A_{O_{IUT}} - outs_{\#s'+1} \wedge s' \in \mathcal{T}(P_S)$

$\Rightarrow$ [$o \in A_{O_{IUT}} - outs_{\#s'+1} \Rightarrow o \in A_{O_{IUT}} \wedge o \notin outs_{\#s'+1}$]
$\exists\, s \mid s \in \mathcal{T}(P_{IUT}) \wedge s = s' \frown \langle o \rangle \wedge o \in A_{O_{IUT}} \wedge o \notin outs_{\#s'+1} \wedge s' \in \mathcal{T}(P_S)$

$\equiv$ [def. $outs_i$]
$\exists\, s \mid s \in \mathcal{T}(P_{IUT}) \wedge s = s' \frown \langle o \rangle \wedge o \in A_{O_{IUT}} \wedge o \notin out(P_S, s') \wedge s' \in \mathcal{T}(P_S)$

$\Rightarrow$ [predicate logics]
$\exists\, s' : \mathcal{T}(P_S) \bullet s' \frown \langle o \rangle \in \mathcal{T}(P_{IUT}) \wedge o \in A_{O_{IUT}} \wedge o \notin out(S, s')$

$=$ [definitions $initials(.)$ and P/s]
$\exists\, s' : \mathcal{T}(P_S) \bullet o \in initials(P_{IUT}/s') \wedge o \in A_{O_{IUT}} \wedge o \notin out(S, s')$

$=$ [definition $out(.)$]
$\exists\, s' : \mathcal{T}(P_S) \bullet o \in out(IUT, s') \wedge o \notin out(S, s')$

$=$ [$\subseteq$-definition]
$\exists\, s' : \mathcal{T}(P_S) \bullet out(IUT, s') \nsubseteq out(S, s')$

$=$ [predicate logics]
$\neg \forall\, s' : \mathcal{T}(P_S) \bullet out(IUT, s') \subseteq out(S, s')$

$=$ [Definition 5]
$\neg\, (IUT\ \mathbf{cspio}\ S)$ ◇

References

[88089] ISO 8807:1989 (1989) LOTOS: a formal description technique based on the temporal ordering of observational behaviour. ISO

[AHKV98] Alur R, Henzinger TA, Kupferman O, Vardi MY (1998) Alternating refinement relations

[And07] Andrade W et al. (2007) Interruption test case generation for mobile phone applications (in Portuguese). In: XXV Brazilian symposium in computer networks and distributed systems

[Ber07] Bertolino A (2007) Software testing research: achievements, challenges, dreams. In: FOSE '07: 2007 future of software engineering. IEEE Computer Society, Washington, pp 85–103

[Bez11] Bezerra R (2011) Extração Automática de Modelos CSP a partir de Casos de Uso. Master's thesis, CIn-UFPE (Center of Informatics of Federal University of Pernambuco)

[BG03] Bertolino A, Gnesi S (2003) Use case-based testing of product lines. SIGSOFT Softw Eng Notes 28(5):355–358

[BJK+05] Broy M, Jonsson B, Katoen J-P, Leucker M, Pretschner A (2005) Model-based testing of reactive systems: advanced lectures (LNCS). Springer-Verlag New York, Inc., Secaucus

[BL02] Briand L, Labiche Y (2002) A UML-based approach to system testing. Softw Syst Model 1(1):10–42

[BMAF10] Bertolini C, Mota A, Aranha E, Ferraz C (2010) GUI testing techniques evaluation by designed experiments. In: International conference on software testing verification and validation (ICST). IEEE, pp 235–244

[CANM08] Cartaxo EG, Andrade WL, Oliveira Neto FG, Machado PDL (2008) LTS-BT: a tool to generate and select functional test cases for embedded systems. In: SAC '08: Proceedings of the 2008 ACM symposium on applied computing. ACM, New York, pp 1540–1544

[CG07] Cavalcanti A, Gaudel M-C (2007) Testing for refinement in CSP. In: ICFEM. LNCS, vol 4789. Springer, Berlin, pp 151–170

[CH09] Colvin R, Hayes IJ (2009) CSP with hierarchical state. In: IFM '09: Proceedings of the 7th international conference on integrated formal methods. Springer, Berlin, pp 118–135

[CJRZ01] Clarke D, Jéron T, Rusu V, Zinovieva E (2001) STG: a tool for generating symbolic test programs and oracles from operational specifications. SIGSOFT Softw Eng Notes 26(5):301–302

[CS08] Cabral G, Sampaio A (2008) Automated formal specification generation and refinement from requirement documents. J Braz Comput Soc 14(1):87–106

[CT08] Cabral G, Tamai T (2008) Requirement-based testing through formal methods. In: Proceedings of TESTCOM-FATES 2008—short papers

[DFM09] Damasceno A, Farias A, Mota A (2009) A mechanized strategy for safe abstraction of CSP specifications. In: Formal methods: foundations and applications. LNCS, vol 5902. Springer, Berlin, pp 118–133

[dPF08] dos Prazeres Farias J (2008) NLScripts: composição assistida de scripts de testes a partir de descrições em linguagem natural controlada (in Portuguese). Master's thesis, CIn-UFPE (Center of Informatics of Federal University of Pernambuco), June 2008

[EEC+04] Edmund SC, Edmund SC, Clarke EM, Sharygina N, Sinha N (2004) State/event-based software model checking. In: Integr Form Methods 2999:128–147

[FNSB10] Ferreira F, Neves L, Silva M, Borba P (2010) TaRGeT: a model based product line testing tool. In: Proceedings of CBSoft 2010—tools panel

[For05] Formal Systems (2005) Failures-divergence refinement—FDR2 user manual. Formal Systems (Europe) Ltd, June 2005

[For11] Formal Systems (2011) Formal systems Web site. http://fsel.com. Sep 2011

[Goi10] Gois FNB (2010) Test script diagram—Um modelo para geração de scripts de testes (in Portuguese). Master's thesis, Universidade de Fortaleza (UNIFOR)

[Gro07] Object Group (2007) OMG unified modeling language (OMG UML). Superstructure, V2.1.2. Technical report

[HBB+09] Hierons R, Bogdanov K, Bowen J, Cleaveland R, Derrick J, Dick J, Gheorghe M, Harman M, Kapoor K, Krause P, Lüttgen G, Simons A, Vilkomir S, Woodward M, Zedan H (2009) Using formal specifications to support testing. ACM Comput Surv 41(2):1–76

[Hen88] Hennessy M (1988) Algebraic theory of processes. MIT Press, Cambridge

[HVFR05] Hartmann J, Vieira M, Foster H, Ruder A (2005) A UML-based approach to system testing. Innov Syst Softw Eng 1(1):12–24

[JJ05] Jard C, Jéron T (2005) 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):297–315

[LdBB+01] Ledru Y, du Bousquet L, Bontron P, Maury O, Oriat C, Potet M-L (2001) Test purposes: adapting the notion of specification to testing. In: Proceedings of IEEE/ACM international conference on automated software engineering, 2001 (ASE 2001). IEEE Computer Society, pp 127–134, Nov 2001

[Lei07] Leitão D et al. (2007) NLForSpec: translating natural language descriptions into formal test case specifications. In: SEKE. Knowledge Systems Institute Graduate School, pp 129–134

[LISA09] Lima L, Iyoda J, Sampaio A, Aranha E (2009) Test case prioritization based on data reuse an experimental study. In: ESEM '09: Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, Washington, DC, USA. IEEE Computer Society, pp 279–290

[MBS02] Mota A, Borba P, Sampaio A (2002) Mechanical abstraction of CSP-Z processes. In: FME 2002: formal methods—getting IT right. LNCS, vol 2391. Springer, Berlin, pp 163–183, Jan 2002

[Mil89] Milner R (1989) Communication and concurrency. Prentice Hall

[MMYS09] Mafra J, Miranda B, Yoda J, Sampaio A (2009) Test case selector: a tool for the selection of test cases (in Portuguese). In: Proceedings of SAST09, Gramado

[NCT+07] Nogueira S, Cartaxo EG, Torres D, Aranha E, Marques R (2007) Model based test generation: an industrial experience. In: SAST 2007—1st Brazilian workshop on systematic and automated software testing, João Pessoa

[NFLTJ06] Nebut C, Fleurey F, Le Traon Y, Jezequel JM (2006) Automatic test generation: a use case driven approach. IEEE Trans Softw Eng 32(3):140–155

[NS11] Nayak A, Samanta D (2011) Synthesis of test scenarios using UML activity diagrams. Softw Syst Model 10:63–89. doi:10.1007/s10270-009-0133-4

[NSM08] Nogueira S, Sampaio A, Mota A (2008) Guided test generation from CSP models. In: Proceedings of the 5th ICTAC. Springer, Berlin, pp 258–273

[NSM11] Nogueira S, Sampaio A, Mota A (2011) Guided test generation from CSP models. Technical report, CIn-UFPE. http://www.cin.ufpe.br/~scn/reports/TR-ATG.pdf

[OB88] Ostrand TJ, Balcer MJ (1988) The category-partition method for specifying and generating fuctional tests. Commun ACM 31(6):676–686

[PS97] Peleska J, Siegel M (1997) Test automation of safety-critical reactive systems. S Afr Comput J 19:53–77

[PY96] Parashkevov AN, Yantchev J (1996) ARC—a tool for efficient refinement and equivalence checking for CSP. In: IEEE international conference on algorithms and architectures for parallel processing (ICA3PP'96), pp 68–75

[RBJ00] Rusu V, du Bousquet L, Jéron T (2000) An approach to symbolic test generation. In: IFM '00: Proceedings of the second international conference on integrated formal methods. Springer, London, pp 338–357

[RG99] Ryser J, Glinz M (1999) A scenario-based approach to validating and testing software systems using statecharts. In: 12th International conference on software and systems engineering and their applications (ICSSEA'99)

[RH07] Roscoe AW, David Hopkins (2007) SVA: a tool for analysing shared-variable programmes. In: Proceedings of AVoCS 2007, pp 177–183

[Ros98] Roscoe AW (1998) The theory and practice of concurrency. Prentice Hall

[Ros11] Roscoe AW (2011) Understanding concurrent system. Springer, Berlin

[Sam05] Sampaio A et al. (2005) Software test program: a software residency experience. In: Proceedings of the 27th ICSE'05 (Education & training track). ACM Press, pp 611–612

[SC08] Somé SS, Cheng X (2008) An approach for supporting system-level test scenarios generation from textual use cases. In: SAC '08: Proceedings of the 2008 ACM symposium on applied computing. ACM, New York, pp 724–729

[Sca98] Scattergood JB (1998) The semantics and implementation of machine-readable CSP. PhD thesis, Oxford University Computing Laboratory

[Sch99] Schneider S (1999) Abstraction and testing. In: FM '99, World Congress on formal methods, vol I. Springer, Berlin, pp 738–757

[SLD08] Sun J, Liu Y, Dong JS (2008) Model checking CSP revisited: introducing a process analysis toolkit. In: Proceedings of the third international symposium on leveraging applications of formal methods, verification and validation (ISoLA 2008). Communications in computer and information science, vol 17. Springer, Berlin, pp 307–322

[SLDC09] Sun J, Liu Y, Dong JS, Chen C (2009) Integrating specification and programs for system modeling and verification. In: Proceedings of the 2009 third IEEE international symposium on theoretical aspects of software engineering, TASE '09, Washington, DC, USA. IEEE Computer Society, pp 127–135

[SNM09] Sampaio A, Nogueira S, Mota A (2009) Compositional verification of input-output conformance via CSP refinement checking. In: ICFEM '09: Proceedings of the 11th international conference on formal engineering methods (invited paper). Springer, Berlin, pp 20–48

[Som06] Somé SS (2006) Supporting use case based requirements engineering. Inf Softw Technol 48(1):43–58

[TB99] Tretmans J, Belinfante A (1999) Automatic testing with formal methods. EuroSTAR'99: 7th European international conference on software testing, analysis & review, Nov 8–12, 1999, pp 8–12

[Tor06] Torres D et al. (2006) Motorola SpecNL: a hybrid system to generate NL descriptions from test case specifications. In: International conference on hybrid intelligent systems, Los Alamitos, CA, USA. IEEE Computer Society, p 45

[Tre96] Tretmans J (1996) Test generation with inputs, outputs and repetitive quiescence. Softw Concepts Tools 17(3):103–120

[Tre99] Tretmans J (1999) Testing concurrent systems: a formal approach. In: Baeten JCM, Mauw S (eds) CONCUR'99. LNCS, vol 1664. Springer, Berlin, pp 46–65

[UPL06] Utting M, Pretschner A, Legeard B (2006) A taxonomy of model-based testing. In: Working paper series. University of Waikato, Department of Computer Science. April 2006

[VCG+08] Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L (2008) Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons R, Bowen J, Harman M (eds) Formal methods and testing. Lecture notes in computer science, vol 4949. Springer, Berlin, pp 39–76

[WP99] Williams C, Paradkar A (1999) Efficient regression testing of multi-panel systems. In: ISSRE '99: Proceedings of the 10th international symposium on software reliability engineering, Washington, DC, USA. IEEE Computer Society, p 158

[WW08] Weiglhofer M, Wotawa F (2008) "On the fly" input output conformance verification. In: Proceedings of the IASTED international conference on software engineering, Innsbruck, Austria. ACTA Press, pp 286–291

Received 12 October 2011

Revised 3 February 2012

Accepted 27 June 2012 by Dong Jin Song

Published online 2 August 2012