Test selection for traces refinement

Ana Cavalcanti

University of York, Department of Computer Science
York YO10 5GH, UK

Marie-Claude Gaudel

LRI, Université de Paris-Sud and CNRS
Orsay 91405, France

Abstract

Theories for model-based testing identify exhaustive test sets: typically infinite sets of tests whose execution establishes the conformance relation of interest. Practical techniques rely on selection strategies to identify finite subsets of these tests, and popular approaches are based on requirements to cover the model. In previous work, we have defined testing theories for refinement-based process algebra, namely, CSP and *Circus*, a state-rich process algebra. In this paper, we consider the selection of tests designed to establish traces refinement. In this case, conformance does not require that all traces of the model are available in the system under test, and this can raise challenges regarding coverage criteria for selection. To address these difficulties, we present a framework for formalising a variety of selection strategies. We exemplify its use in the formalisation of a selection criterion based on coverage of process communications for integration testing. We consider models written in *Circus*, whose symbolic testing theory facilitates the definition of uniformity and regularity hypotheses based on data operations, but also imposes extra challenges for selection of concrete tests. Our results, however, are relevant for any formalism where the conformance relation does not require all the traces of the specification to be executable by the system under test.

Keywords: process algebra, CSP, *Circus*, synchronisation coverage, integration testing

1. Introduction

We address in this paper the issue of selecting tests when using an abstract specification for testing that a system behaves like one of its refinements. In particular, we are interested in models written using process algebra like CSP [51] or *Circus* [17], and in traces refinement in that context.

CSP is a well established notation that has been in use for more than twenty years. The availability of a powerful model checker has led to acceptance both in academia and industry. In the public domain, we have reports on applications in hardware and e-commerce [2, 26], for example. CSP has been combined with a number of data-modelling languages to define notations that cope with state-rich reactive systems [21, 36, 52, 49]; *Circus* is one of these combinations with Z [62]. *Circus* has been used for modelling and verification of control systems specified in Simulink [9, 38]. It is currently being used to verify aerospace applications, including virtualisation software by the US Naval Research Laboratory [23].

The traditional area of application for CSP and *Circus* is refinement-based development and verification, not testing, but whenever a CSP or a *Circus* model is available, the possibility of using it for testing is attractive, especially in industry. Moreover, when the correctness criterion for development is refinement, it is natural to adopt that same criterion for testing. We have previously developed testing theories for CSP [10] and *Circus* [13]. We base our discussions here on the theory for *Circus*, which extends and generalises that for CSP. In addition, it enables richer selection strategies based on data types (as well as traces).

The various notions of refinement adopted in the context of CSP and *Circus* require that the set of traces of an implementation is included in that of the specification; in particular, this is exactly the definition of traces refinement. Selection of tests for traces refinement is challenging because it is acceptable that some traces of the specification are not implemented by the system under test (SUT). It may, therefore, be the case that some selected tests are not executable. As a consequence, given a selection criterion, the selection cannot be performed statically only. At runtime, the test driver must ensure that alternative tests are submitted until the criterion is fulfilled, or until it is certain that it cannot, due to lack of traces necessary to its fulfillment. In other words, any testing strategy requires a combination of some selection criterion with a driver designed to ensure that the selection criterion will be reached at run time, if it is possible.

Our main contributions are as follows. First, we propose a formal framework for selection based on the definition of an exhaustive set of tests that can contribute to the fulfillment of the selection criterion and of a test driver. Additionally, we consider symbolic tests, which pose a challenge of the same nature for obtaining concrete tests that can be used by the driver, since it is acceptable that some or all instantiations are not implemented. Symbolic tests lead to a two-level selection method, where first some symbolic tests are selected, and then we select some of their instantiations, which are submitted until the criterion is fulfilled. Finally, as an application of our new framework, we consider coverage of communications between parallel processes of a *Circus* model; we present a complete formalisation of this selection criterion.

Our work bears some similarity with the dynamic test generation techniques [6], where symbolic execution is used at runtime to ensure coverage of all feasible paths of a program text. Here, however, the issue is not infeasibility, but arbitrary partial implementation of traces. Moreover, we are concerned with selection criteria that are based on requirements to exercise elements of the model, not the SUT. Our framework also has some similarities with adaptive testing [33], which is aimed at tackling nondeterminism in implementations rather than partial implementation of traces. In contrast, for coverage of behaviours of a nondeterministic SUT, we rely on the complete testing assumption [24], which may, in special cases, be enforced by techniques similar to those used in reachability testing [34, 39].

In the next section, we give an overview of *Circus* and its testing theory. Section 3 presents our novel framework for selection. In Section 4, we formalise a synchronisation coverage criterion for use with our framework, and we define finite test sets and suitable drivers. We consider related works in Section 5, and conclude in Section 6 with a discussion of our results and future work.

2. *Circus* and its testing theory

In this section, we cover the background material for our work: we describe *Circus* using an example of a simple protocol, and introduce the main concepts of its testing theory.

2.1. An overview of Circus

In Figure 1, we present a *Circus* model for a simple protocol for communication between a *Sender* and a *Receiver* via a *Medium* that can corrupt at most one out of three messages. Systems and their components are modelled in *Circus* using processes. These encapsulate some state and interact with each other and their environment via channel communications, which are synchronous, atomic, and instantaneous events. In Figure 1, we have a model that uses four channels, *in*, *out*, *left*, and *right*, and introduces five processes: *Medium*, *Sender*, *Receiver*, *Components*, and *Protocol*.

We observe that, although *Circus* events (channel communications) are synchronous, this example shows that its modelling paradigm is rich enough to cater for asynchronous systems. Our example models an asynchronous communication between the *Sender* and the *Receiver* by identifying the relevant interaction points of the asynchronous transaction (using the channels *left* and *right*).

A *Circus* model is formed by a sequence of paragraphs. In Figure 1, the first paragraph defines a set *Bit* including 0 and 1. The messages exchanged using the specified protocol are bits. In the second paragraph, we declare the channels, whose type *Bit* defines that they are used to communicate bits.

The next three paragraphs in Figure 1 introduce (basic) processes by (explicitly) defining their states and behaviour. Process states are defined using Z schemas. The state of the process *Medium* has a single

$$Bit == \{0, 1\}$$

$$\textbf{channel}\ in, out, left, right : Bit$$

$$\textbf{process}\ Medium \mathrel{\widehat=} \textbf{begin}$$
$$\quad \textbf{state}\ MS == [\, noE : 0 \mathinner{..} 2 \,]$$
$$\quad \bullet\ \ noE := 0 \,;\, \left(\mu\, X \bullet \left(\begin{array}{l} left?x \longrightarrow \left(\begin{array}{l} \textbf{if}\ noE = 0 \longrightarrow \\ \quad (right!x \longrightarrow \textbf{Skip} \sqcap right!(1 - x) \longrightarrow noE := 2) \\ [\!]\ noE > 0 \longrightarrow right!x \longrightarrow noE := noE - 1 \\ \textbf{fi} \end{array} \right) \end{array} \right) \right) ;\ X \Bigg)$$
$$\textbf{end}$$

$$\textbf{process}\ Sender \mathrel{\widehat=} \textbf{begin}$$
$$\bullet\ (\mu\, X \bullet in?m \longrightarrow left!m \longrightarrow left!m \longrightarrow left!m \longrightarrow X)$$
$$\textbf{end}$$

$$\textbf{process}\ Receiver \mathrel{\widehat=} \textbf{begin}$$
$$\quad \textbf{state}\ MS == [\, ms : \mathrm{bag}\ Bit \,]$$
$$\quad \bullet\ \ \left(\mu\, X \bullet \left(\begin{array}{l} ms := [\![\ ]\!]\ ;\ right?m1 \longrightarrow right?m2 \longrightarrow right?m3 \longrightarrow \\ ms := [\![m1, m2, m3]\!]\ ;\ out!(maj\ ms) \longrightarrow X \end{array} \right) \right)$$
$$\textbf{end}$$

$$\textbf{process}\ Components \mathrel{\widehat=} Sender \,|||\, Receiver$$

$$\textbf{process}\ Protocol \mathrel{\widehat=} (Components \,[\![\, \{\!|\, left, right \,|\!\} \,]\!]\, Medium) \setminus \{\!|\, left, right \,|\!\}$$

Figure 1: *Circus* example: a simple protocol

component *noE* recording the number of messages that must be sent correctly before another error is possible. We use this information to model the assumption that at most one out of every three messages is corrupted. The declaration of *noE* indicates that it can only take the values 0, 1, and 2.

After the $\bullet$, we have a *Circus* action that specifies the behaviour of *Medium*. It is written using a combination of CSP operators and guarded commands. First of all, an assignment initialises *noE* to 0, so that a corrupt message can be sent immediately. Afterwards, a recursive action iteratively accepts messages through the channel *left* and sends them through *right*. Recursion is defined using the operator $\mu X \bullet A(X)$, which introduces the local name X for the action A, in which references to X are recursive calls.

In the body of the recursion in *Medium*, we have an input communication $left?x$ that accepts an input through the channel *left* and records it in the local variable x. The prefixing operator $c \longrightarrow A$ describes a behaviour in which, after the communication c, we have the execution of the action A. In *Medium*, after the communication $left?x$, we have a conditional that checks the value of *noE*. If it is 0, then it is possible that a corrupt message is sent. In this case, we have a nondeterministic choice between outputting through the channel *right* the input x (communication $right!x$) or the corrupt value $1 - x$, in which case the value 2 is assigned to *noE*. The nondeterministic choice operator $A_1 \sqcap A_2$ chooses arbitrarily between the execution of A_1 or A_2. In our example, the nondeterminism reflects the fact that we are abstracting from the possible causes of a medium error, so that the occurrence of an error becomes completely arbitrary.

If the action chosen is $right!x \longrightarrow \textbf{Skip}$, then after the output over *right*, it behaves like **Skip**, which terminates immediately without changing the state. If the other action is chosen, after the output, the value of *noE* is updated to indicate that the next two communications must not be corrupt. In the conditional, if the value of *noE* is not 0, then $right!x$ takes place, and the value of *noE* is decremented.

The process *Sender* has no state. Its action defines that it accepts an input m through the channel *in* and sends it three times to the *Medium* through the channel *left*, before recursing and starting again.

The *Receiver* process, on the other hand, keeps a bag of bits *ms* in the state. At each iteration of the

recursion in its action, the bag is emptied ($ms := [\![\ ]\!]$), three messages m_1, m_2, and m_3 are accepted (from the *Medium*) through *right*. These messages are stored in *ms*, and then the bit (*maj ms*) with the majority of occurrences in *ms* is output through *out*. (The definition of *maj ms* is omitted in Figure 1.)

Circus processes can also be combined using CSP operators. In our example, the components connected by the protocol are modelled by the process *Components*, defined by the interleaving of the processes *Sender* and *Receiver*. This reflects the fact that these processes are executed in parallel and independently. The *Protocol* process itself is defined in terms of the parallel composition of *Components* and *Medium*. In this case, we have a parallelism, rather than an interleaving, because these processes need to synchronise on communications over the channels *left* and *right*. In a parallelism $P_1 \ [\![\ cs\ ]\!]\ P_2$, the processes P_1 and P_2 proceed independently, except only for communications on channels in the given set *cs*, on which they are required to synchronise. In particular, the state of P_1 is not visible to P_2, and vice-versa.

In the definition of *Protocol*, we also use hiding. In a process $P \setminus cs$ the communications over the channels in the set *cs* are hidden: they are internal and carried out as soon as possible. In our example, *left* and *right* are channels used by the *Medium* to connect the *Components*, and so they are internal. Communication with the external environment is realised via the channels *in* (used in *Sender*) and *out* (used in *Receiver*).

We use this and other examples to illustrate our results. Any extra use of *Circus* notation is explained.

The semantics of *Circus* is defined in a relational style using Hoare and He's Unifying Theories of Programming (UTP) [43]; operational semantics for *Circus* are defined in [13, 12]. In [12] we define traces refinement using the UTP model. As expected, the definition compares processes P_1 and P_2 with the same alphabet of events, and requires that P_1 is refined by P_2 if, and only if, every trace of P_2 is also a trace of P_1. In the next section, we give an overview of the *Circus* theory for testing for traces refinement.

2.2. Testing against traces refinement using Circus (and CSP)

In a theory for model-based testing for a particular modelling notation and conformance relation, we identify testability hypotheses, notions of test, test execution, and verdict, and an exhaustive test set. This is a set of tests whose execution and associated verdicts can determine unequivocally whether the SUT behaves in conformance to a given model, as long as the testing hypotheses hold. Typically, this is an infinite set of tests. Here, we consider selection as the definition of a subset of the exhaustive test set. Thus, we briefly recall the definition of the exhaustive test set for *Circus* and traces refinement [13].

The two testability hypotheses of the *Circus* theory are standard. First, we assume that the SUT can be described by an (unknown) *Circus* model. This means, for example, that events are atomic (indivisible) and can be considered as instantaneous: their duration can be neglected when observed at the level of the system. Secondly, we rely on the complete test assumption [24]. It is concerned with possible nondeterminism in the SUT (as opposed to its model); it requires the existence of a number n such that, if a test is executed n times, then all possible (nondeterministic) behaviours of the SUT are observed. For a deterministic SUT, n is 1. When the only source of nondeterminism is due to interleaving of concurrent sequential executions, the complete test assumption can be enforced by the use of reachability-testing techniques to monitor the executions of the SUT. Some of these techniques are briefly presented in Section 5.

The tests of the *Circus* theory embed the verdict. We use special verdict events *inc*, *pass*, and *fail* used to indicate that a test is *inc*onclusive, because the SUT did not execute the particular trace of events attempted by the test, or that the SUT *pass*ed or *fail*ed the test. A test is constructed based on a particular trace t of the model, and a forbidden continuation e of t. A forbidden continuation is an event that is not allowed after the trace t; in other words, the trace $t \frown \langle e \rangle$ is not a trace of the model. (We use $\langle e \rangle$ to denote the singleton sequence containing just the event e, and $\frown$ is the concatenation operator.)

Example 1. *In our example, $\langle in.0, out.0, in.1 \rangle$ is a trace of Protocol. The event $in.0$, for instance, corresponds to a value 0 input through the channel in, and, similarly, $out.0$, the value 0 output through the channel out. After the trace $\langle in.0, out.0, in.1 \rangle$, all events $out.x$, where $x \neq 1$ are forbidden continuations of Protocol, because after an input of a value 1, just 1 itself can be output.* □

A test for a trace t and a forbidden continuation e attempts to drive the SUT to execute t. If it does not succeed, the test is inconclusive. (As usual, we give an inconclusive verdict when the intended test is

not completed.) If the test does succeed in driving the SUT to execute t, then the forbidden event e is attempted, and if it is accepted, the SUT fails the test. Otherwise, it succeeds.

Example 2. *For the trace in Example 1, and forbidden continuation out.2, we have the test below.*

$$inc \longrightarrow in.0 \longrightarrow inc \longrightarrow out.0 \longrightarrow inc \longrightarrow in.1 \longrightarrow pass \longrightarrow out.2 \longrightarrow fail \longrightarrow \mathbf{Stop}$$

The action **Stop** *deadlocks immediately.* □

The verdict events are interspersed with the events of the trace and the forbidden continuation. The last verdict event before a deadlock determines the verdict of the test. Formally, tests are defined as follows.

Definition 1.

$$T_T(\langle\,\rangle, e) = pass \rightarrow e \rightarrow fail \rightarrow \mathbf{Stop} \qquad T_T(\langle\, e_1 \,\rangle \frown t, e_2) = inc \rightarrow e_1 \rightarrow T_T(t, e_2)$$

In $T_T(t, e)$, extending the trace t with e is supposed to lead to an invalid trace, since e is supposed to be a forbidden continuation: not an initial after t; as already explained, the test aims at ruling it out.

Execution $Execution^{SP}_{SUT}(T)$ of one of these tests T is carried out by executing T and the SUT in parallel, synchronising on all model events (that is, the non-verdict events used in the model and in the SUT). Relying on our first testability hypothesis, we use SUT to denote the unknown model of the SUT. We also use Σ to denote the set of model events as defined in the specification SP.

Definition 2.

$$Execution^{SP}_{SUT}(T) = (SUT \,[\![\, \Sigma \,]\!]\, T) \setminus \Sigma \quad where \quad \Sigma = \alpha SP$$

The model events are hidden, that is, internal to the experimentation process. This ensures that they are not affected by the environment of the experiment, since they are not visible in the environment. Hiding also ensures that these model events occur as soon as they become available (in the SUT and in the test).

For a given model SP, the exhaustive test set $Exhaust_T(SP)$ for traces refinement is built by considering all traces of SP and all their forbidden continuations. We consider the set $traces(SP)$ [11], which characterises a trace semantics for SP. We also consider the set $initials(SP, t)$ of continuations of the trace t of SP.

Definition 3 (Exhaustive test set of SP for traces refinement). *Given a specification SP, we define $Exhaust_T(SP) = \{\, T_T(t, e) \mid t \in traces(SP) \wedge e \notin initials(SP, t) \,\}$.*

Exhaustiveness of $Exhaust_T(SP)$ is formally established by the following theorem, proved in [10].

Theorem 1 (Exhaustivity of $Exhaust_T$). *Given two processes, SUT and SP, $SP \sqsubseteq_T SUT$ if, and only if $\forall\, T_T(s, a) : Exhaust_T(SP);\ t : traces(Execution^{SP}_{SUT}(T_T(s, a))) \bullet last(t) \neq fail$*

This extreme test strategy requires the execution of all the traces of SP with all their forbidden continuations. It yields an inconclusive verdict when a tested trace is not implemented. If there are only *pass* and *inc* verdicts, it guarantees that the SUT refines SP providing our testability hypotheses hold.

As said above, when selecting a strict subset of $Exhaust_T$, and running the tests T of this set as described by $Execution^{SP}_{SUT}(T)$ an element to be covered can be statically covered in the subset, but not covered at runtime. In Section 3, we introduce the notions of exhaustive coverage of an element of the specification, and runtime coverage of this element by the implementation. Before discussing these novel concepts, however, we discuss briefly a symbolic account of the *Circus* tests.

2.3. Symbolic tests in Circus

The testing theory described above was first cast in the context of CSP [10], rather than *Circus*. The main difference between *Circus* and CSP, however, is the added possibility of specifying complex data types using the predicative style of pre and postconditions. Since *Circus* processes encapsulate the data, it is not surprising that tests can be constructed from a *Circus* process in the same way as from a CSP process. With similar notions of traces and trace refinement, all results of the CSP theory are maintained.

On the other hand, it is possible and, of interest, to take into account the data operations of a model when specifying a testing strategy. To this end, we have developed a symbolic characterisation of tests and exhaustive test sets for *Circus* [13]. It is based on the *Circus* operational semantics, which maintains a symbolic account of the state of a process. We give a brief description of the main definitions here.

The *Circus* operational semantics defines a transition relation $\mathtt{p}_1 \xrightarrow{\mathtt{l}} \mathtt{p}_2$ between texts $\mathtt{p}_1$ and $\mathtt{p}_2$ of processes. When $\mathtt{p}_1 \xrightarrow{\mathtt{l}} \mathtt{p}_2$ holds, we say that there is a transition from $\mathtt{p}_1$ to $\mathtt{p}_2$ with label $\mathtt{l}$. We use the typewriter font to distinguish texts (of processes, actions, predicates, and so on), from the entities that they denote. If the label is ϵ, the execution of the process p_1 can evolve silently (without any interaction with the environment), so that we then have an execution of p_2. On the other hand, there may be a label $\mathtt{d.w}$, representing a communication over the channel d of a value represented by the symbolic variable w. In this case, the execution of p_1 evolves to that of p_2, after engaging in the communication $d.w$.

The definition of this transition relation for processes is based on a transition relation for configurations involving actions. They are triples $(\mathtt{c} \mid \mathtt{s} \models \mathtt{A})$ where $\mathtt{c}$ is a constraint over the symbolic variables in use, $\mathtt{s}$ is an assignment to the *Circus* variables in scope of values represented by symbolic variables, and $\mathtt{A}$ is the text of a *Circus* action. The symbolic variables constrained in $\mathtt{c}$ are those used in labels $\mathtt{d.w}$ to represent communications and those used in the definition of the state $\mathtt{s}$ to represent values of *Circus* variables.

To construct symbolic tests, we use the notion of constrained symbolic traces of a process. A constrained symbolic trace is a pair formed by a symbolic trace $\mathtt{st}$ and a constraint $\mathtt{c}$. A symbolic trace is a finite sequence of symbolic events $\mathtt{d}.\alpha_0$, where d is a channel, and α_0 is a symbolic variable that represents the value communicated. The constraint $\mathtt{c}$ is the text of a predicate over the symbolic variables used in $\mathtt{st}$.

Example 3. *For the Protocol process, we have the constrained symbolic trace* $(\langle in.\alpha, out.\beta, in.\gamma\rangle, \alpha = \beta)$*, for example. It represents all traces where the first input and output values* α *and* β *are the same, and then a new, unconstrained value* γ *is input. The trace in Example 1 is one of its instances.* □

We define below the function *cstraces* that characterises the set of constrained symbolic traces of a process. It is parametrised by an alphabet $\mathtt{a}$, which fixes the symbolic variables that can be used in the symbolic trace, and the order in which they can be used. For a process $\mathtt{begin\ state}\ [\mathtt{x} : \mathtt{T}] \bullet \mathtt{A}\ \mathtt{end}$, the constrained symbolic traces are those of its main action $\mathtt{A}$, starting from a state in which the state x takes the value w_0 constrained by $w_0 \in T$. This is the set $cstraces^a(\mathtt{w}_0 \in \mathtt{T}, \mathtt{x} := \mathtt{w}_0, \mathtt{A})$ defined using the operational semantics.

Definition 4.

$$cstraces^{\mathtt{a}}(\mathtt{begin\ state}[\mathtt{x} : \mathtt{T}] \bullet \mathtt{A}\ \mathtt{end}) = cstraces^{\mathtt{a}}(\mathtt{w}_0 \in \mathtt{T}, \mathtt{x} := \mathtt{w}_0, \mathtt{A})$$

$$cstraces^{\mathtt{a}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1) = \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \\ \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \end{array} \right\}$$

In defining $cstraces^a(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1)$, we consider the configurations $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ that can be reached by evaluation of the operational semantics from the configuration $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$ identified by the parameters. They are characterised by the transition relation $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$, which is annotated with a trace $\mathtt{st}$. This transition relation is defined in terms of the operational semantics in the usual way.

The symbolic trace $\mathtt{st}$ records the events that arise from the evaluation of $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$ in reaching $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ as illustrated in Example 3. With $\alpha\mathtt{st} \leq \mathtt{a}$, we require that the symbolic variables used to define $\mathtt{st}$, like α, β and γ in Example 3, are those of the alphabet $\mathtt{a}$, in the order in which they appear in $\mathtt{a}$. We use $\alpha\mathtt{st}$ to refer to the sequence of variables used in $\mathtt{st}$, and $\alpha\mathtt{st} \leq \mathtt{a}$ requires that it is a prefix of $\mathtt{a}$.

The constraint $\mathtt{c}_2$ associated with $\mathtt{st}$, however, includes constraints on symbolic variables that are not used in $\mathtt{st}$, but are used to represent internal values arising from the evaluation of $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$. These symbolic variables should not be mentioned in the constrained symbolic trace. Above, we use the function $\mathcal{R}$ defined below to remove them. When applied to a trace $(\mathtt{st}, \mathtt{c})$, it quantifies away every variable in the set $\alpha\mathtt{c}$ of free variables of $\mathtt{c}$ that is not in the set $\alpha\mathtt{st}$ of symbolic variables used in $\mathtt{st}$.

Definition 5. *For every* $(\mathtt{st}, \mathtt{c})$*, we define* $\mathcal{R}(\mathtt{st}, \mathtt{c})$ *as follows.*

$$\mathcal{R}(\mathtt{st}, \mathtt{c}) = (\mathtt{st}, \exists(\alpha\mathtt{c} \setminus \alpha\mathtt{st}) \bullet \mathtt{c})$$

The operational semantics guarantees that all configurations have satisfiable constraints. Therefore, every

constrained symbolic trace is feasible, in the sense that it has at least one valid instance, and so it is guaranteed that the constraint $\mathtt{c}_2$ to which $\mathcal{R}$ is applied in Definition 4 is satisfiable.

As said above, to construct tests, we also need forbidden continuations. In the symbolic approach, we use symbolic events: pairs formed by a symbolic event and a constraint. A symbolic event is itself formed by a channel name and a symbolic variable. We define the set $\overline{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ of constrained symbolic events over the alphabet $\mathtt{a}$ that represent the events that are not initials of $\mathtt{P}$ for any of the instances of its symbolic trace $(\mathtt{st}, \mathtt{c})$. A definition of this set is in [13], and its use for test generation is detailed in [20].

Example 4. *The set* $\overline{csinitials}^{\mathtt{a}}(\mathtt{Protocol}, (\langle \mathtt{in}.\alpha, \mathtt{out}.\beta, \mathtt{in}.\gamma \rangle, \alpha = \beta))$ *of constrained symbolic events for Protocol after the trace in Example 3, where* $\mathtt{a}$ *is the alphabet* $\langle \alpha, \beta, \gamma, \delta, \ldots \rangle$*, contains the symbolic event* $(out.\delta, \alpha = \beta \wedge \gamma \neq \delta)$*. It determines that the value* δ *output after an input* $\mathtt{in}.\gamma$ *is different from* γ*. We observe that the constraint is over the variable used in the symbolic event, as well as the variables used in the symbolic trace that lead to it. In this example, the constraint refers to* α*,* β*, and* γ*, as well as* δ*.* □

From the symbolic traces and events, we can construct symbolic tests in the expected way. For a constrained symbolic trace $(\mathtt{st}, \mathtt{c}_1)$ and a symbolic event $(\mathtt{d}.\beta_0, \mathtt{c}_2)$ that represents a forbidden continuation, the corresponding symbolic test is given by $\mathtt{ST}^{\alpha}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\beta_0, \mathtt{c}_2))$ defined below.

Definition 6.

$$\mathtt{ST}^{\alpha}_{\mathtt{T}}((\langle\,\rangle, \mathtt{c}_1), (\mathtt{d}.\beta_0, \mathtt{c}_2)) = \mathtt{pass} \longrightarrow \mathtt{d}?\beta_0 : \mathtt{c}_2 \longrightarrow \mathtt{fail} \longrightarrow \mathtt{Stop}$$

$$\mathtt{ST}^{\alpha}_{\mathtt{T}}((\langle\, \mathtt{d}_{\mathtt{st}}.\alpha_0 \,\rangle \frown \mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\beta_0, \mathtt{c}_2)) = \mathtt{inc} \longrightarrow \mathtt{d}_{\mathtt{st}}?\alpha_0 : (\exists\, \overline{\alpha, \alpha_0} \bullet \mathtt{c}_1) \longrightarrow \mathtt{ST}^{(\alpha, \alpha_0)}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\beta_0, \mathtt{c}_2))$$

For a constrained symbolic trace $(\langle\,\rangle, c_1)$ whose symbolic trace is empty, we only add to the test the verdict event *pass*. After that, corresponding to a symbolic forbidden continuation $(d.\beta_0, c_2)$, we have an input communication $d?\beta_0 : c_2$, which accepts an input β_0 through the channel d, with the constraint $\mathtt{c}_2$ restricting the accepted values of the variable β_0. Similarly, for the prefixings corresponding to each of the symbolic events $\mathtt{d}_{\mathtt{st}}.\alpha_0$ of the symbolic trace $\mathtt{st}$, we use the constraint $\mathtt{c}_1$ to extract the right constraint over the input α_0. The α parameter of $\mathtt{ST}_{\mathtt{T}}$ records the symbolic variables already used, and the constraint on α_0 is obtained by quantifying all variables not in α and different from α_0, namely, those in the list $\overline{\alpha, \alpha_0}$.

The instantiation of a symbolic test is defined below as a function that yields a set of concrete tests.

Definition 7.

$$\begin{aligned}
&\mathit{instTest}(\mathtt{Stop}) = \{\, \mathbf{Stop} \,\} \\
&\mathit{instTest}(\mathtt{d} \longrightarrow \mathtt{A}) = \{\, TA : \mathit{instTest}(\mathtt{A}) \bullet d \longrightarrow TA \,\} \\
&\mathit{instTest}(\mathtt{d}?\alpha_0 : \mathtt{c} \longrightarrow \mathtt{A}) = \{\, v_0,\, TA \mid c[v_0/\alpha_0] \wedge TA \in \mathit{instTest}(\mathtt{A}[\mathtt{v}_0/\alpha_0]) \bullet d.v_0 \longrightarrow TA \,\}
\end{aligned}$$

The instances of a symbolic test are built in the following way. The action $\mathtt{Stop}$ is not really symbolic; its only instance is itself. For a prefixing $\mathtt{d} \longrightarrow \mathtt{A}$, whose communication is a simple synchronisation like $\mathtt{inc}$, $\mathtt{pass}$, or $\mathtt{fail}$, for example, the instances are prefixings formed out of d itself and the instances TA of the prefixed action $\mathtt{A}$. Finally, if we have a prefixing $\mathtt{d}?\alpha_0 : \mathtt{c} \longrightarrow \mathtt{A}$, then the instances consider all possible ways of choosing for communication a value v_0 that satisfies the constraint c, and the corresponding instances of the action $\mathtt{A}[\mathtt{v}_0/\alpha_0]$, where the choice of v_0 for α_0 is recorded.

Example 5. *For the constrained symbolic trace and forbidden continuation in Example 4, we get the following symbolic test.*

$$\begin{aligned}
&\mathtt{inc} \longrightarrow \mathtt{in}.\alpha : \mathtt{true} \longrightarrow \mathtt{inc} \longrightarrow \mathtt{out}.\beta : \alpha = \beta \longrightarrow \mathtt{inc} \longrightarrow \mathtt{in}.\gamma : \alpha = \beta \longrightarrow \\
&\mathtt{pass} \longrightarrow \mathtt{out}.\delta : \alpha = \beta \wedge \gamma \neq \delta \longrightarrow \mathtt{fail} \longrightarrow \mathbf{Stop}
\end{aligned}$$

The test in Example 2 is an instance of this test. □

A slightly different definition of instantiation that avoids the enumeration of all the instances of the final forbidden symbolic events is possible: given a symbolic test constructed from the constrained symbolic trace $(\mathtt{st}, \mathtt{c}_1)$ followed by the forbidden constrained symbolic event $(\mathtt{d}.\beta_0, \mathtt{c}_2)$, the variables in $(\mathtt{st}, \mathtt{c}_1)$ are instantiated with values that satisfy the constraint $\mathtt{c}_1$, and these instantiations are propagated to $\mathtt{c}_2$ as described above, but the final constraint is not instantiated. In this case, instances of symbolic tests are *Circus* processes where the single constrained symbolic event in the symbolic test is used in a final synchronisation with a predicate. Such an instance of the symbolic test of Example 5 is as follows.

$$\mathtt{inc} \longrightarrow \mathtt{in.0} \longrightarrow \mathtt{inc} \longrightarrow \mathtt{out.0} \longrightarrow \mathtt{inc} \longrightarrow \mathtt{in.1} \longrightarrow \mathtt{pass} \longrightarrow \mathtt{out}.\delta : \mathtt{1} \neq \delta \longrightarrow \mathtt{fail} \longrightarrow \mathtt{Stop}$$

A symbolic exhaustive test set is just a set of symbolic tests. Selection strategies based on symbolic tests are the subject of Section 3, selection of instances is addressed in Section 3.4, and further selection focused on coverage of synchronisations in *Circus* model is studied in Section 4.5.

3. Selection and traces refinement

The criteria for selecting a subset of the generally infinite set of possible tests are the core of any testing strategy. This can be achieved via coverage criteria, test purposes, uniformity or regularity hypotheses, or more generally some property of tests [28]. In the testing theory for CSP and *Circus*, tests are built from traces of the specification and a forbidden continuation; the exhaustive test set covers all these traces and events. So, selection must ultimately define a subset of the set of traces and their forbidden continuations.

As already mentioned, when the considered conformance relation is traces refinement, it is acceptable that a trace of the specification is not implemented in the SUT. As seen in Section 2.2, in this case execution of the corresponding test yields an *inc*onclusive verdict. Thus it may be the case that a selected test is not executable, and if there is another test that may be able to fulfill the selection criterion, it must be executed, and we should proceed in this way, until either a satisfactory test is executed or it is certain that none exists. This has an impact on the way in which test selection for traces refinement can be defined and applied.

There are two main interrelated tasks involved in the definition of a selection strategy for traces refinement: a (formal) definition of (a) the (possibly still infinite) subset of all those tests that can contribute to the fulfillment of the selection criterion and (b) a test driver. We explain in the next section why defining an exhaustive test set with respect to the selection criteria is useful when considering traces refinement.

Whether concrete or symbolic tests are considered is irrelevant for definition of the exhaustive test set. We just observe that the notion of constrained symbolic traces, which is central in the *Circus* testing theory, is well suited for the definition of uniformity subdomains in the selection of both traces and initials: the constraints of the symbolic traces and events can be reused. On the other hand, we need to construct a driver of concrete tests. When selection is based on symbolic tests, we have, in addition, to address the fact that it may be the case that not all instantiations of a selected symbolic test are executable by the SUT.

Next, we illustrate the problem of selection for traces refinement. In Section 3.2, we consider several approaches to the definition of an exhaustive test set and of a test driver with respect to various selection criteria. In Section 3.3, we address factorisation of tests as for adaptive testing [33], and briefly discuss online and offline test generation. Section 3.4 discusses symbolic tests. Finally, Section 3.5 considers the issues related to the finiteness of the exhaustive test sets with respect to the given criteria.

3.1. The problem

To explain the problem associated with test selection for traces refinement in detail, we consider the case where the selection is based on a coverage criterion, that is, a set of elements of the specification are required to be exercised by at least one test experiment. The elements may be, for example, synchronisation events, pairs of related events, or more sophisticated requirements on execution traces.

If the test set is selected in a static way, it may turn out that the test selected to ensure coverage of a given element reaches an inconclusive verdict before executing it because the trace is not implemented. It may be the case, however, that other traces that provide coverage are implemented and that covering it is achievable in the SUT. Thus it is necessary to select and design tests that try at runtime other ways of covering this element, in order to ensure that if it is executable by the SUT, then it is exercised.

Example 6. *For simplicity, we consider channels a, b, c, d and e that are used for synchronisation, but do not communicate any value. In this case, the constrained symbolic traces degenerate to standard traces, because there is no need to introduce symbolic variables to represent communicated values, and the only valid constraint is just* `true`*. In addition, we consider a stateless Circus process P defined below.*

$$P \mathrel{\widehat=} \mathbf{begin} \bullet a \rightarrow e \rightarrow c \rightarrow \mathbf{Skip} \sqcap b \rightarrow e \rightarrow d \rightarrow \mathbf{Skip}\ \mathbf{end}$$

In what follows, whenever we present a stateless process like this, for brevity, we omit the **begin** *and* **end** *keywords and identify the process with its action after the '$\bullet$'.*

Some of the processes that trace-refine P are P itself and processes whose main actions are either of the actions in the internal choice. A test set that covers the event e, for instance, is the singleton including the process whose action is $\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$*. To be sure, however, that if a trace covering e exists in the SUT, then e is covered by some test experiment, we need to identify the set of all tests that provide the required coverage and use a new way of driving these tests. Alternatively, a new form of test may be used, which adapts at runtime to the SUT.*

The first solution is to design a driver based on a set of all tests covering e that will trigger their executions until e is exercised. The second solution is to have tree-shaped tests such as:

$$\texttt{inc} \longrightarrow \left(\begin{array}{l} \texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \\ \Box \\ \texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{c} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \end{array} \right)$$

This test offers to the SUT the choice between two traces of the specification that pass through e. With this test, if the SUT is deterministic, if there exists in the SUT a specified trace exercising e, it is executed. When the SUT is not deterministic, by applying the test a sufficient number of times, under the complete test assumption, we know that the trace is executed. If there is no such trace, the test is inconclusive. □

In the next section we explore both solutions indicated in the above example.

3.2. A solution: monitoring runtime coverage

As said above, when combining test selection and traces refinement, use of a selection criterion requires both the selection of traces and forbidden continuations and the design of a driver that ensures that the criterion is fulfilled at runtime, if possible. We proceed by selecting all the tests that satisfy the criterion, and define the corresponding driver that tries all these tests if necessary until a test is successfully completed.

Example 7. *For illustration, we consider as selection criterion coverage of a particular event e. In this case, given a specification SP, the exhaustive set of tests that satisfy the selection criterion is as follows.*

$$Exhaust_T(SP)\restriction_e = \{\, T_T(t, a) \mid t \in traces(SP) \land t\restriction_e \neq \langle\rangle \land a \notin initials(SP, t) \,\}$$

where $t\restriction_e$ eliminates from t all events different from e. This subset of $Exhaust_T(SP)$ covers all the traces where e occurs at least once. For Example 6, the tests in this set are as follows.

$\texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{a} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{b} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{inc} \longrightarrow \texttt{c} \longrightarrow \texttt{pass} \longrightarrow \texttt{a} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{inc} \longrightarrow \texttt{c} \longrightarrow \texttt{pass} \longrightarrow \texttt{b} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
...
$\texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{a} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{b} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{c} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
$\texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{inc} \longrightarrow \texttt{d} \longrightarrow \texttt{pass} \longrightarrow \texttt{a} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$
...

We define below a test driver that runs all these tests until a pass verdict, possibly followed by a fail verdict

is observed. This is ensured by a test monitor process TMonitor executed alongside the tests.

$$Driver(SP, SUT)\restriction_e = (|||\ T : Exhaust_T(SP) \restriction_e \bullet Execution^{SP}_{SUT}(T)) \ [\![\{\!| inc, pass, fail |\!\}]\!]\ TMonitor$$

where

$$TMonitor = inc \rightarrow TMonitor \,\Box\, pass \rightarrow fail \rightarrow \mathbf{Stop}$$

Running $Driver(SP, SUT)\restriction_e$ ensures that if e is executable by the SUT, it is covered at runtime. This means that some test based on a trace where e occurs reaches a conclusive verdict: pass or fail. (If the SUT is nondeterministic, the driver needs to be executed several times, and we need to rely on the complete test assumption.) The tests are executed in interleaving (in parallel independently) until one of them reaches a pass event. In this case, we either have an immediate deadlock, or just the associated forbidden continuation as an internal event of the test execution, followed by an observable fail. We recall that all events of the specification are internal to a test execution, and so not visible to the TMonitor process.

A shortcoming of the above driver is that it makes an implicit arbitrary selection of a forbidden continuation. A more liberal TMonitor should allow all (or specific) forbidden continuations to be tested.

If e is not covered at runtime with the strategy above, all the tests return an inconclusive verdict. On the other hand, the fact that all tests return an inconclusive verdict does not mean that e has not been executed since some of the tests may return an inconclusive verdict after having exercised e. □

Selection of tests is often based on a given property π of traces or tests. Examples of such properties are coverage of an event, as considered above, or absence of a particular subsequence of events, and so on. In this case, it is usual to consider as selection criteria either: at least one test satisfying π is executed, or all the tests satisfying π are executed. We call the former existential selection of tests with respect to π, and the latter universal selection with respect to π. We can imagine, of course, some intermediate criteria where several of these tests are selected, or some different criterion that corresponds to a global property of the set of selected tests. As explained later on, however, these selection criteria can be addressed using compositions and variations of the definitions we describe below.

Properties of traces, rather than tests, may also be considered for selection criteria. In what follows, we first discuss existential or universal selection of traces and, in this case, the fact that there are several tests for one trace must be taken into account. Afterwards, we discuss selection based on properties of tests, and consider existential and universal selection as informally discussed above. We observe that the problem discussed in the previous section, which arises from non-implemented traces of the specification, is only an issue for existential selection (of either traces of tests) and its variants.

To characterise the criteria based on properties of traces, we note that for each trace t of a specification SP, we have the set of tests $AllTests_T(t, SP)$ defined below.

$$AllTests_T(t, SP) = \{\ T_T(t, e) \mid e \notin initials(SP, t)\ \}$$

It contains all tests built from t and one of its forbidden continuations e.

For a property π of traces, we have the possible selection criteria below.

existential selection of traces: there is *at least one* trace t satisfying π, such that, $AllTests_T(t, SP)$, the test set identified by t is executed (that is, all tests in $AllTests_T(t, SP)$ are executed); or

universal selection of traces: for *all implemented* traces t satisfying π, $AllTests_T(t, SP)$ is executed.

Moreover, several properties $\pi_1, \ldots, \pi_n$ of traces may be considered; the coverage of a set of elements is an example. In such cases, one may require either of the following criteria.

multiple existential selection of traces: for every π_i, there is at least one trace t satisfying π_i, such that $AllTests_T(t, SP)$ is actually executed; or

multiple universal selection of traces: for every π_i, for every implemented trace t satisfying π_i, the set $AllTests_T(p, SP)$ is executed.

Coming back to test selection, for a given property π of tests, one may require either of the following.

existential selection of tests: there is *at least one* test T satisfying π, that is actually executed, if there is one that is implemented, or

universal selection of tests: *all implemented* tests T satisfying π are executed.

Direct test selection makes it possible to focus on specific forbidden continuations, for instance to consider only some specific forbidden event. The property π may also combine requirements on the traces and the forbidden continuations, like in, for instance, "all the tests with a trace t where event $e1$ occurs and event $e2$ is the forbidden continuation". In Example 7, for instance, such a criterion for test selection allows us to restrict the set of relevant tests to those that have `a`, for example, as a forbidden continuation. In this case, only three of the tests enumerated in Example 7 are retained for testing. Multiple existential and universal selection criteria for tests can be tackled in the same way as those for traces. Below, we consider exhaustive test sets and drivers for each of the above classes of traces and tests selection criteria.

3.2.1. Existential selection of traces

For existential selection, the selected test set $Existential_T^{trace}(SP)\restriction_\pi$ needs to be structured as a set of test sets characterised by $AllTests_T$, as defined below.

$$Existential_T^{trace}(SP)\restriction_\pi = \{AllTests_T(t, SP) \mid t \in traces(SP) \wedge t \vdash \pi\} \quad (1)$$

We use the notation $t \vdash \pi$ to indicate that the trace t satisfies the property π.

The execution of all the tests associated with an implemented trace t is specified below as a simple driver that runs independently and to conclusion all tests in $AllTests_T(t, SP)$.

$$|||\ T : AllTests_T(t, SP) \bullet Execution_{SUT}^{SP}(T)$$

A driver that ensures existential coverage can be specified as follows: independently, for each trace t that defines a set of tests $AllTests_T(t, SP)$ in $ExistentialS(SP)\restriction_\pi$, the driver above is run under the control of a monitor that observes whether a *pass* event takes place. When it does, it means that t is implemented. The monitor continues the execution of the other tests based on t, and sends to the other drivers for the other traces a *done* event to make them to stop. Formally, we have the following definition for a generic driver, for a specification SP and some property π of a trace.

$$\begin{aligned} &ExistDriver^{trace}(SP, SUT)\restriction_\pi = \\ &\quad \Big([\![\{\!| done |\!\}]\!]\ TS : Existential_T^{trace}(SP)\restriction_\pi \bullet \\ &\qquad (|||\ T : TS \bullet Execution_{SUT}^{SP}(T)\ )\ [\![\{\!| inc, pass, fail |\!\}]\!]\ TSMonitor \Big) \setminus \{\!| done |\!\} \end{aligned}$$

where *TSMonitor* is below. It stops when a *done* occurs, and raises a *done* event when a *pass* occurs.

$$TSMonitor = done \rightarrow \texttt{Stop} \,\Box\, inc \rightarrow TSMonitor \,\Box\, pass \rightarrow done \rightarrow Run$$

Run just continues running the other tests based on the same trace.

$$Run = inc \rightarrow Run \,\Box\, pass \rightarrow Run \,\Box\, fail \rightarrow Run$$

In the definition above of $ExistDriver^{trace}(SP)\restriction_\pi$, each of the sets TS in $Existential_T^{trace}(SP)\restriction_\pi$ is considered. The drivers for each of these sets are run in parallel, synchronising on the event *done*. Each driver runs all tests T in TS independently, but under the control of *TSMonitor*, which observes *done* as well as the verdict events. If a *done* event occurs, *TSMonitor* stops (and so the tests under its control stop), because this indicates that a monitor for another test set has observed a test with a conclusive verdict. Otherwise, *TSMonitor* and its tests proceed until a *pass* is observed, when *done* is raised, but the controlled tests proceed to conclusion; *Run* just ignores all verdict events. The *done* event is local to the existential driver; it is used only for synchronisation between the monitors of the tests associated with each trace t. Termination of the driver depends of the size of the set of selected tests. This is further discussed in Section 3.5.

Example 8. *Coming back to coverage of an event e considered in Example 7, we have:*

$$Existential_T^{trace}(SP)\restriction_{(t\restriction_e \neq \langle\rangle)} = \{AllTests_T(t, SP) \mid t \in traces(SP) \wedge t \restriction e \neq \langle\rangle\}$$

The distributed union of the test sets in this set is exactly $Exhaust_T(SP)\restriction_e$. Here, their grouping based on the traces t as defined by $AllTests_T(t, SP)$ allows us to consider the alternative driver below.

$$\begin{array}{l} ExistDriver^{trace}(SP, SUT)\restriction_{(t\restriction_e \neq \langle\rangle)} = \\ \quad \Big([\![\{\!| done |\!\}]\!] \; TS : Existential_T^{trace}(SP) \restriction_{(t\restriction_e \neq \langle\rangle)} \bullet \\ \qquad (\;|||\; T : TS \bullet Execution_{SUT}^{SP}(T)\;) \; [\![\{\!| inc, pass, fail |\!\}]\!] \; TSMonitor \Big) \setminus \{\!| done |\!\} \end{array}$$

This driver ensures that for at least one implemented trace covering e, if there is one, all the tests are executed, that is, all the forbidden events are attempted. As already said, the driver $Driver(SP, SUT) \restriction e$ presented in Example 7 attempts only one of them chosen arbitrarily. □

As usual, if the SUT is nondeterministic, coverage has to rely on the complete test assumption and on running the driver several times. This is the case for all drivers that we present.

3.2.2. Universal selection of traces

The case of universal selection is easier to formalise; there is no need to consider a set of test sets as in the previous section. The test set to be considered is as follows.

$$Universal_T^{trace}(SP)\restriction_\pi = \{\, T_T(t, a) \mid t \in traces(SP) \wedge t \vdash \pi \wedge a \notin initials(SP, t) \,\} \tag{2}$$

All tests that are implemented must be run, so the driver attempts to execute all tests.

$$UnivDriver^{trace}(SP, SUT)\restriction_\pi \;=\; |||\; T : Universal_T^{trace}(SP) \restriction_\pi \bullet Execution_{SUT}^{SP}(T)$$

Example 9. *Coming back again to coverage of an event e, we have*

$$Exhaust_T(SP)\restriction_e = Universal_T^{trace}(SP)\restriction_{(t\restriction_e \neq \langle\rangle)}$$

The corresponding driver $UnivDriver^{trace}(SP, SUT)\restriction_{(s\restriction_e \neq \langle\rangle)}$ executes all tests arising from all traces that include at least one occurrence of e. The driver $Driver(SP, SUT)$ in Example 7 executes only one of these tests for each trace with at least one occurrence of e. □

3.2.3. Multiple existential or universal selection of traces

Selection criteria based on a set of properties $\Pi = \{\pi_1, \ldots, \pi_n\}$ can be handled by considering the test sets and the drivers corresponding to each property π_i. For multiple existential coverage of traces, we have a set of test sets for each π_i, and the driver executes independently each of the existential drivers.

$$MExistDriver^{trace}(SP, SUT)\restriction_\Pi \;=\; |||\; \pi_i : \Pi \bullet ExistDriver^{trace}(SP, SUT)\restriction_{\pi_i}$$

Example 10. *A classical example of a multiple existential selection criteria is that every event must be covered at least once. For a set E of events, we have the following driver:*

$$MExistDriver^{trace}(SP, SUT) \restriction E \;=\; |||\; e : E \bullet ExistDriver^{trace}(SP, SUT)\restriction_e$$

□

For multiple universal selection criteria, we can consider a test set containing all tests obtained for each property π_i. There is no need to keep separate sets of tests, since they are all to be executed. We can, therefore, use a driver similar to $UnivDriver^{trace}(SP, SUT)\restriction_\pi$ previously presented.

3.2.4. Existential or universal selection of tests

For a given property π of tests, we consider, for both existential and universal selection, the following exhaustive test set. We use $T \vdash \pi$ to denote the fact that the test T satisfies π.

$$Exhaust_T^{test}(SP)\restriction_\pi = \{ T : Exhaust_T(SP) \mid T \vdash \pi \}$$

There is no need to use a set of test sets like we have done for existential selection of traces.

For existential selection of the tests satisfying π, the driver is as follows.

$$ExistDriver^{test}(SP, SUT)\restriction_\pi = \\ (|||\ T : Exhaust_T^{test}(SP)\restriction_\pi \bullet Execution_{SUT}^{SP}(T)) \llbracket \{\!| inc, pass, fail |\!\} \rrbracket TMonitor$$

where *TMonitor* is as in Example 7.

$$TMonitor = inc \rightarrow TMonitor \,\Box\, pass \rightarrow fail \rightarrow \mathbf{Stop}$$

For universal selection of tests satisfying π, the driver is shown below.

$$UnivDriver^{test}(SP, SUT)\restriction_\pi \ = \ |||\ T : Exhaust_T^{test}(SP)\restriction_\pi \bullet Execution_{SUT}^{SP}(T)$$

This is exactly the same driver used for universal selection based on traces, except that it uses the tests in $Exhaust_T^{test}(SP)\restriction_\pi$ (rather than those in $Exhaust_T^{trace}(SP)\restriction_\pi$).

To give an example, we define below the process $end_T(T)$; it captures the final behaviour of a test T, where, after signalling a *pass*, it attempts to execute a forbidden continuation.

Definition 8.

$$end_T(pass \longrightarrow P) = P \qquad end_T(inc \longrightarrow e \longrightarrow P) = end_T(P)$$

This definition is used in the next example and in the next section.

Example 11. *One may consider the universal selection criterion "all the tests where a given event e is a forbidden continuation", possibly because it is especially critical not to execute this event e when it is forbidden by the specification. In this case, the test set is as follows.*

$$Exhaust_T^{test}(SP)\restriction_{\neg e} = \{ T : Exhaust_T(SP) \mid end(T) = e \rightarrow fail \rightarrow \mathbf{Stop} \}$$

□

More imaginative selection criteria can be dealt with by the definition of adequate test sets or sets of test sets, or by composition of drivers. We have considered the main kinds of selection strategies, and shown how to construct the corresponding exhaustive test set for the required selection criteria and accompanying driver. We cannot claim to have explored every conceivable criterion, but note that driver composition with exchanges of messages, as done for existential selection, can cater for quite a wide range of selection strategies. If needed, it is possible to specify in *Circus* drivers that use, for example, counters and guards.

3.3. Factorisation

As already said, instead of using a monitor process to define special drivers, it is possible to factorise the tests in the exhaustive test set to define a single tree-shaped test as it is done for adaptive testing in [33]. This test offers the choice between all the tests selected; the second test in Example 6 illustrates the idea.

Factorisation decreases the number of inconclusive verdicts, since it gives the SUT the choice of several continuations available in the test set. Factorised tests have, however, a difficulty concerning coverage of behaviours, since the choices are no more under the control of the test and can be biased by the SUT.

Example 12. *We consider, for instance, the factorisation of the following tests.*

$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$
$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{c} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$

It gives the following tree-shaped test.

$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \left(\begin{array}{l} \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \\ \Box \\ \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{c} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \end{array} \right)$$

After the SUT performs the trace $\langle\, a \,\rangle$, there is a choice between the testing events pass and inc that is left up not to the test, but to the environment of the test execution. It is an external choice over events that are not in the alphabet of the SUT. □

For the non-factorised tests, the environment of a test execution can be a simple process that is prepared to accept interaction on any of the events *inc*, *pass*, and *fail*, at any moment. Even in such a liberal environment, it is guaranteed that the interaction with the SUT defined by the trace that corresponds to the test is attempted in the test execution. If the test is factorised, though, the environment of the test execution is offered choices between *pass* and *inc* events, whenever there is a possibility, according to the specification, of extending the trace or deadlocking. To ensure coverage, both choices should be tried.

A related concern arises when there are several possibilities of failure.

Example 13. *We consider, for instance, the factorisation of the following tests.*

$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$
$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{pass} \longrightarrow \texttt{e} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$

It gives the following tree-shaped test.

$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{pass} \longrightarrow \left(\begin{array}{l} \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \\ \Box \\ \texttt{e} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop} \end{array} \right)$$

In this case, after the occurrence of a, if the environment of the test execution chooses to synchronise on pass, it then has no control over the choice between d and e: these events are not in its alphabet. The test is not making a choice, but rather offering it to the SUT. If the SUT can perform any of them, the mistake is going to be detected, but it is not possible to determine which of the mistaken events are possible, since the SUT is not forced to perform any of them. □

It remains, however, that the SUT drives the test by making the (external) choices of model events offered in the test. Since just one path of the tree-shaped test is executed in a test experiment, we have an existential selection (by the SUT) of a test combined into the tree. Moreover, only the complete test assumption and enough executions of the tree-shaped test can guarantee that all paths in the tree-shaped test (and, therefore, all tests combined in the tree) are executed.

We define below a function $Fact^{SP}(TS)$ that defines the factorised tree-shaped tests coming from a test set TS. It is defined in terms of a function $FactP^{SP}(TS, t)$, which takes as an extra parameter the path (a trace) t that defines the branch of the tree-shaped test that is being constructed.

Definition 9 (Factorisation).

$$Fact^{SP}(TS) = FactP^{SP}(TS, \langle\,\rangle)$$

$$FactP^{SP}(TS, t) = TFail(TS, t) \;\Box\; TCont^{SP}(TS, t)$$

$$TFail(TS, t) = \mathbf{Stop} \qquad [\text{if } t \notin trace \llparenthesis\, TS \,\rrparenthesis]$$

$$TFail(TS, t) = pass \longrightarrow \Box\ T : \{\, T : TS \mid trace_T(T) = t \,\} \bullet end_T(T) \qquad [\text{if } t \in trace \llparenthesis\, TS \,\rrparenthesis]$$

$$TCont^{SP}(TS, t) = \mathbf{Stop} \qquad [\text{if } initials_T(TS, t) = \varnothing]$$

$$TCont^{SP}(TS, t) = inc \longrightarrow \Box\ e : initials_T(TS, t) \bullet e \longrightarrow FactP^{SP}(TS, t \frown \langle e \rangle) \qquad [\text{if } initials_T(TS, t) \neq \varnothing]$$

For any set TS, $Fact^{SP}(TS)$ is given by $FactP^{SP}(TS, \langle\,\rangle)$; initially, just the empty path has been constructed. For any path t, this is an external choice between tests defined by functions $TFail(TS, t)$ and $TCont^{SP}(TS, t)$. With $TFail(TS, t)$, we consider whether t is the trace of a test in TS or not. We define the set $trace \llparenthesis\, TS \,\rrparenthesis$ of traces of a set of tests TS as the set of specification traces used in its tests, not including the forbidden continuations. We use the relational image operator $f \llparenthesis\, S \,\rrparenthesis$; it provides the set of results obtained by applying f to each element in the set S. The trace of a single test T is defined as follows.

Definition 10.

$$trace_T(pass \longrightarrow P) = \langle\,\rangle \qquad trace_T(inc \longrightarrow e \longrightarrow P) = \langle e \rangle \frown trace_T(P)$$

If t is not a trace of a test in TS, then $TFail(TS, t)$ is **Stop**, which is a unit for external choice: $\mathbf{Stop} \Box P = P$. For example, TS may not have any test for the empty trace $\langle\,\rangle$, and in this case $TFail(TS, \langle\,\rangle)$ does not add anything to the tree-shaped test. On the other hand, if there are tests T in TS whose trace is t, then, for each of them, we add its treatment of the forbidden continuation, characterised by $end_T(T)$, to the tree.

With $TCont^{SP}(TS, t)$, we consider whether t is the prefix of a trace of a test in TS or not. We use the notion of initials of a test set TS after a trace t formally defined as follows.

Definition 11.

$$initials_T(TS, t) = \{\, e \mid \exists\, T : TS;\ s \bullet trace_T(T) = t \frown \langle e \rangle \frown s \,\}$$

If t has no continuation, then $TCont^{SP}(TS, t)$ is just **Stop**. Otherwise, we add an intermediary verdict *inc* followed by a branch (in an external choice) for each of the continuations e.

Example 14. *The test in Example 6 can be built from the test set including*

$$\texttt{inc} \longrightarrow \texttt{a} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{d} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$

and

$$\texttt{inc} \longrightarrow \texttt{b} \longrightarrow \texttt{inc} \longrightarrow \texttt{e} \longrightarrow \texttt{pass} \longrightarrow \texttt{c} \longrightarrow \texttt{fail} \longrightarrow \texttt{Stop}$$

No special driver is required in this case. □

As already indicated above, however, the two approaches discussed here, namely, factorisation and the definition of exhaustive test sets and drivers, are not equivalent. In the execution of a factorised test, the control, exercised via choice of events, is left to the SUT or the test environment. This is in contrast with the drivers in Section 3.2, which control the execution of the tests to which they are associated.

A third approach is to alternate the control between the driver and the SUT, as in on-the-fly testing [58, 59, 32], where test derivation and execution are integrated into a single process. This approach is also called online testing, in contrast with offline testing, where test generation is a separate process that takes places before execution. Online testing is especially relevant when testing reactive systems: the next event of the tester is generated in function of the reaction of the SUT.

In this paper, we have not addressed the issue of generation of the various exhaustive test sets occurring in the drivers: they are defined in an abstract, predicative way. Similarly, the factorisation function above starts from a given subset of the exhaustive test set, and thus is not a generation algorithm. The corresponding

test-generation algorithms from a *Circus* specification can be developed and used either offline, following the patterns of the drivers defined in Section 3.2, or online.

Online testing would avoid running in parallel all the tests that satisfy the criterion. On the other hand, online testing would require integrating in the driver the generation algorithm of the exhaustive test sets for ensuring that all the relevant tests are attempted until one or all are completed (depending of the kind of considered selection: existential or universal).

3.4. Partial implementation of symbolic tests

As explained in Section 2.3, the *Circus* testing theory offers a symbolic characterisation of the concrete tests and of the exhaustive concrete test set. It is possible to specify a testing strategy for a *Circus* specification at the concrete level, just following what is presented in Section 3.2 or 3.3. Uniformity hypotheses on instantiations of constrained symbolic tests, however, provide a natural selection criterion. For instance, a simple hypothesis states that the verdict for all instances of a symbolic test is the same. In this case, we assume that the verdict of the test in Example 2 is the same verdict of all instances of the symbolic test in Example 5. Constrained symbolic tests enable the definition of powerful selection strategies that take into account data types properties. We, therefore, consider now how to select from symbolic tests, and then instantiate the selected tests and run them via concrete drivers to ensure runtime coverage.

We have a two-level selection method. Firstly, we need to provide a symbolic exhaustive test set with respect to the selection criteria of interest. Just like in Section 3.2, this amounts to the definition of all those symbolic tests that, when instantiated, may contribute at runtime to the satisfaction of the criterion. In this case, we need to rely on properties of symbolic traces or symbolic tests, but that is all.

Secondly, we need to define drivers for the instantiations of the symbolic tests. The challenge is that, when starting the testing experiments with a given symbolic test, it may be only partially implemented. Moreover, if there is no implemented instantiation, it is necessary to try another symbolic test of the exhaustive test set, and actually all of them until either one concrete test reaches a conclusion, or it is certain that it is impossible to fulfill the criterion for the SUT. Mechanisms similar to those in Section 3.2 must be used for existential or universal selection among those concrete tests corresponding to a symbolic test.

The execution of all the implemented instances of a given symbolic test ST is specified as follows.

$$UnivInstDriver(ST, SP, SUT) = \Big|\Big|\Big| \; T : instTest(ST) \bullet Execution^{SP}_{SUT}(T)$$

As said above, however, symbolic tests are generally used to select only one of their instantiations. The execution of at least one implemented instance of a symbolic test ST, if there is one, is specified below.

$$ExistInstDriver(ST, SP, SUT) = \\ (\Big|\Big|\Big| \; T : instTest(ST) \bullet Execution^{SP}_{SUT}(T)) \; [\![\; \{\!| inc, pass, fail |\!\} \;]\!] \; TMonitor$$

This is similar to the existential selection driver addressed in Section 3.2, but we monitor tests coming from a single test set, namely, the set of instantiations of the symbolic test ST, rather than a set of tests. As soon as a test T in the set $instTest(ST)$ reaches a *pass* verdict, the monitor stops all other tests. The test that reached a *pass* is allowed to conclude (by possibly giving a later *fail* verdict) before stopping.

The two specifications above can be realised by the use at runtime of some adequate solver for enumerating instantiations in a way similar to [56] and [3]. In this case, the instances are generated one at a time, until a *pass* event is observed; only the instances of the same symbolic test that generated the current concrete test are then considered, and no other symbolic tests are tried.

In summary, in a selection strategy for symbolic tests, we can have existential or universal selection of symbolic traces or tests, with an existential or universal selection of instances of symbolic tests. In combining the results in Section 3.2 with $UnivInstDriver(ST, SP, SUT)$ or $ExistInstDriver(ST, SP, SUT)$ presented above, what we need to do is to use these drivers as the characterisation of the execution of a symbolic test. For example, if we have an existential selection of symbolic traces (based on a property π), the set

$Existential_T^{trace}(SP)\restriction_\pi$ as defined in Section 3.2 is a set of sets of symbolic tests. In this case, if universal selection of instances is required, the existential driver to be used should be as follows.

$$
\begin{array}{l}
ExistUnivDriver^{cstrace}(SP, SUT)\restriction_\pi = \\
\quad \Big([\![\{\!| \mathit{done} |\!\}]\!] \; TS : Existential_T^{trace}(SP) \restriction_\pi \bullet \\
\qquad (\Big|\Big|\Big| \; T : TS \bullet UnivInstDriver(T, SP, SUT) \;) \; [\![\{\!| \mathit{inc}, \mathit{pass}, \mathit{fail} |\!\}]\!] \; TSMonitor \Big) \setminus \{\!| \mathit{done} |\!\}
\end{array}
$$

This is the existential driver from Section 3.2, but the process that captures execution of a test is not $Execution_{SUT}^{SP}(T)$, but the universal driver $UnivInstDriver(T, SP, SUT)$ for symbolic tests defined above. Each TS is a set of symbolic tests for the same symbolic trace. Each T in TS is, therefore, a symbolic test, which is executed by $UnivInstDriver(T, SP, SUT)$. As soon as an instance of a symbolic test reaches a *pass*, just as for concrete tests, the *TSMonitor* stops the tests for (the instances of) symbolic tests in the other sets of symbolic tests. On the other hand, all instances of all other symbolic tests in the same set, which cover all symbolic forbidden continuations, are allowed to proceed to completion.

If, on the other hand, existential selection of instances is required, the driver to be used is as follows.

$$
\begin{array}{l}
ExistExistDriver^{cstrace}(SP, SUT)\restriction_\pi = \\
\quad \Big([\![\{\!| \mathit{done} |\!\}]\!] \; TS : Existential_T^{trace}(SP) \restriction_\pi \bullet \\
\qquad (\Big|\Big|\Big| \; T : TS \bullet ExistInstDriver(T, SP, SUT) \;) \; [\![\{\!| \mathit{inc}, \mathit{pass}, \mathit{fail} |\!\}]\!] \; TSMonitor \Big) \setminus \{\!| \mathit{done} |\!\}
\end{array}
$$

In this case, as soon as an instance of a symbolic test reaches a *pass*, the *TSMonitor* stops the tests for the other sets of symbolic tests. On the other hand, one instance of all other symbolic tests in the same set is allowed to complete. So, for each forbidden continuation, at least one instance is executed.

Example 15. *For illustration, we consider* $Existential_T^{trace}(SP)\restriction_\pi$ *to be a set containing just two sets of symbolic tests. The first set contains tests for the symbolic trace* $(\langle \mathtt{a}.\alpha, \alpha \in \{\mathtt{6},\mathtt{7}\}\rangle$*; we assume that the forbidden continuations for this trace are* $(\mathtt{a}.\beta, \alpha \in \{\mathtt{6},\mathtt{7}\} \wedge \alpha = \beta)$ *and* $(\mathtt{b}, \alpha \in \{\mathtt{6},\mathtt{7}\})$*. The second set of tests is for the empty trace* $(\langle\,\rangle, \mathtt{true})$ *with the single forbidden continuation* $(\mathtt{out}.\alpha, \alpha = \mathtt{5})$*. The first set has two tests, one for each forbidden continuation, and the second has just one symbolic test.*

$$
\begin{array}{l}
Existential_T^{trace}(SP)\restriction_\pi = \\
\quad \{ \; \{ \; \mathtt{inc} \longrightarrow \mathtt{a}.\alpha : \alpha \in \{\mathtt{6},\mathtt{7}\} \longrightarrow \mathtt{pass} \longrightarrow \mathtt{a}.\beta : \alpha \in \{\mathtt{6},\mathtt{7}\} \wedge \alpha = \beta \longrightarrow \mathtt{fail} \longrightarrow \mathtt{Stop}, \\
\qquad \mathtt{inc} \longrightarrow \mathtt{a}.\alpha : \alpha \in \{\mathtt{6},\mathtt{7}\} \longrightarrow \mathtt{pass} \longrightarrow \mathtt{b} : \alpha \in \{\mathtt{6},\mathtt{7}\} \longrightarrow \mathtt{fail} \longrightarrow \mathtt{Stop} \,\}, \\
\quad \{ \mathtt{pass} \longrightarrow \mathtt{out}.\alpha : \alpha = \mathtt{5} \longrightarrow \mathtt{fail} \longrightarrow \mathtt{Stop} \,\} \,\}
\end{array}
$$

Each test in the first set has two instantiations: one where α *(and* β*) takes the value 6, and another where it takes the value 7. There is only one instantiation of the test in the second set. The driver for universal instantiation is the following process as defined by* $ExistUnivDriver^{cstrace}(SP, SUT)\restriction_\pi$*.*

$$
\left(\left(\left(\begin{array}{l}
\left(\begin{array}{l}
Execution_{SUT}^{SP}(\mathit{inc} \longrightarrow a.6 \longrightarrow \mathit{pass} \longrightarrow a.6 \longrightarrow \mathit{fail} \longrightarrow \mathbf{Stop}) \\
|||\\
Execution_{SUT}^{SP}(\mathit{inc} \longrightarrow a.7 \longrightarrow \mathit{pass} \longrightarrow a.7 \longrightarrow \mathit{fail} \longrightarrow \mathbf{Stop}) \\
|||\\
Execution_{SUT}^{SP}(\mathit{inc} \longrightarrow a.6 \longrightarrow \mathit{pass} \longrightarrow b \longrightarrow \mathit{fail} \longrightarrow \mathbf{Stop}) \\
|||\\
Execution_{SUT}^{SP}(\mathit{inc} \longrightarrow a.7 \longrightarrow \mathit{pass} \longrightarrow b \longrightarrow \mathit{fail} \longrightarrow \mathbf{Stop})
\end{array}\right) \\
\quad [\![\{\!| \mathit{inc}, \mathit{pass}, \mathit{fail} |\!\}]\!] \\
\quad TSMonitor
\end{array}\right) \right.\right. \\
\quad [\![\{\!| \mathit{done} |\!\}]\!] \\
\left.\left.\left(\begin{array}{l}
Execution_{SUT}^{SP}(\mathit{pass} \longrightarrow \mathit{out}.5 \longrightarrow \mathit{fail} \longrightarrow \mathbf{Stop}) \\
\quad [\![\{\!| \mathit{inc}, \mathit{pass}, \mathit{fail} |\!\}]\!] \\
TSMonitor
\end{array}\right)\right)\right) \setminus \{\!| \mathit{done} |\!\}
$$

In this case, when one TSMonitor detects a pass event, it stops the other monitor and its tests, but lets all

its other tests run to conclusion. The driver for existential instantiation, on the other hand, is as follows.

$$
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
Execution^{SP}_{SUT}(inc \longrightarrow a.6 \longrightarrow pass \longrightarrow a.6 \longrightarrow fail \longrightarrow \mathbf{Stop}) \\
|||\\
Execution^{SP}_{SUT}(inc \longrightarrow a.7 \longrightarrow pass \longrightarrow a.7 \longrightarrow fail \longrightarrow \mathbf{Stop})
\end{array}\right) \\
[\![\{\!| inc, pass, fail |\!\}]\!] \\
TMonitor
\end{array}\right) \\
|||\\
\left(\begin{array}{l}
\left(\begin{array}{l}
Execution^{SP}_{SUT}(inc \longrightarrow a.6 \longrightarrow pass \longrightarrow b \longrightarrow fail \longrightarrow \mathbf{Stop}) \\
|||\\
Execution^{SP}_{SUT}(inc \longrightarrow a.7 \longrightarrow pass \longrightarrow b \longrightarrow fail \longrightarrow \mathbf{Stop})
\end{array}\right) \\
[\![\{\!| inc, pass, fail |\!\}]\!] \\
TMonitor
\end{array}\right)
\end{array}\right) \\
[\![\{\!| inc, pass, fail |\!\}]\!] \\
TSMonitor
\end{array}\right) \\
[\![\{\!| done |\!\}]\!] \\
\left(\begin{array}{l}
Execution^{SP}_{SUT}(pass \longrightarrow out.5 \longrightarrow fail \longrightarrow \mathbf{Stop}) \\
[\![\{\!| inc, pass, fail |\!\}]\!] \\
TSMonitor
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right)
\setminus \{\!| done |\!\}
$$

In this case, once a TSMonitor detects a pass, it stops the other TSMonitor processes as before, and the TMonitor processes under its control allow one test of each set of instantiations to conclude. □

For universal selection of symbolic tests, again, where $Execution^{SP}_{SUT}(T)$ is used in the driver in Section 3.2, we can use either $UnivInstDriver(T, SP, SUT)$ or $ExistInstDriver(T, SP, SUT)$. Accordingly, we either have all instances or one instance of all symbolic tests run.

If the execution of one instance of one symbolic test is required, what we need is existential selection of symbolic tests, rather than symbolic traces. For the construction of the driver, we proceed as shown above.

We observe that the test drivers presented above can also handle the variant notion of instantiation with symbolic final events defining synchronisations with predicates mentioned at the end of Section 2.3.

3.5. Finiteness

Selection criteria must identify a finite number of tests for execution. The exhaustive test sets for a selection criteria, however, are not necessarily finite.

We have suggested the use of classical uniformity and regularity hypotheses [4]. Uniformity hypotheses are based on the definition of a finite collection of test subsets (uniformity domains), with the assumption that in each subset the tests succeed or fail uniformly. Although there is a finite number of uniformity domains, each may have an infinite number of (symbolic) tests that can contribute to coverage of that domain. In fact, uniformity hypotheses do not eliminate tests from the exhaustive test set; they just group them to form the uniformity domains since they are multiple existential selection criterion (see Section 3.2.3).

Regularity hypotheses are based on a bound of some metrics on the tests: their length, number of occurrences of some events, or number of couples of events, for instance. Typically, a regularity hypothesis eliminates tests from the exhaustive test set. In the presence of an infinite number of events, like in typical *Circus* specifications, regularity hypotheses may not be enough to ensure finiteness. For instance, with an infinite number of events it is possible to construct an infinite number of traces of a given size.

The need for drivers that potentially try all tests in an exhaustive test set, as explained above, can lead to drivers that do not necessarily terminate if the exhaustive test set is infinite. In the next section, we give an example of a sophisticated selection criterion for symbolic tests. We use it in Section 4.5 in the context of our framework, and show how we select finite test sets for that example.

4. Synchronisation coverage in *Circus*

So far, all our examples are based on generic properties of traces or tests for selection. In this section, we consider a more specific selection criterion, based on the coverage of synchronisations between parallel

processes of a *Circus* model. To express this criterion formally, we need to identify traces of the model that might lead to at least one such synchronisation. This is the non-trivial task that we address in this section. We identify the relevant traces of the specification from traces of its parallel components. In addition, we prove that the set that we characterise is indeed a set of traces of the specification and, therefore, adequate for use in the context of our framework to define exhaustive test sets. Our formalisation and proof is in the context of the symbolic operational semantics of *Circus*, and defines how to deal with constraints and alphabets of symbolic variables in constrained symbolic traces.

The need for specific notions of coverage for concurrent programs was recognised very early. The first precise propositions were published at the end of the eighties, when the first general concurrent programming languages appeared (see Section 5 and [55]). More specifically, synchronisation coverage was discussed in [31], where a coverage criterion based on the execution of sequences of communications is proposed, and in [5] where synchronisation coverage for Ada programs is defined and instrumented. More recent work [48, 47] has focussed on integration testing based on coverage of communications between components of distributed real-time systems specified using deterministic timed interface automata. We take inspiration from these works and consider coverage of communications (synchronisations) between parallel processes in *Circus*.

In Section 4.1, we define the set $ScstraceP_{\mathtt{cs}}^{(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1,\mathtt{P}_2)$ of pairs of constrained symbolic traces of parallel processes $\mathtt{P}_1$ and $\mathtt{P}_2$ that can lead to their synchronisation on channels in the set $\mathtt{cs}$. We use this to define the set $Scstraces_{\mathtt{cs}}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1,\mathtt{P}_2)$ of synchronisation traces in Section 4.2. In Section 4.3, we generalise our definitions for parallelisms involving more than two processes. Finally, in Section 4.4, we define the set $selectS^{\mathtt{a}}(P)$ of synchronisation traces of any *Circus* model P. Our main result in this section establishes that this is a subset of the traces of P, as required for the selection strategies discussed in the previous section.

4.1. Synchronisation pairs

We consider parallel processes $\mathtt{P}_1$ and $\mathtt{P}_2$ and their sets of constrained symbolic traces $cstraces^{\mathtt{a}_1}(\mathtt{P}_1)$ and $cstraces^{\mathtt{a}_2}(\mathtt{P}_2)$. We identify below the sets of pairs of traces from each process that can lead to their synchronisation, at least once, when they are executed in parallel with a synchronisation set $\mathtt{cs}$.

Definition 12 (Synchronising constrained symbolic trace pairs). *For processes* $\mathtt{P}_1$ *and* $\mathtt{P}_2$*, synchronisation channel set* $\mathtt{cs}$*, and disjoint alphabets* $\mathtt{a}_1$ *and* $\mathtt{a}_2$*, we define* $ScstraceP_{\mathtt{cs}}^{(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1,\mathtt{P}_2)$ *as follows.*

$$\begin{aligned} ScstraceP_{\mathtt{cs}}^{(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1,\mathtt{P}_2) = \; & \{\, \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2) \mid \\ & \quad (\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \land (\mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \land satisfiable_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2) \\ & \} \end{aligned}$$

A synchronising pair of traces $\mathtt{cst}_1$ and $\mathtt{cst}_2$ is characterised in terms of their projections to communications over channels in $\mathtt{cs}$ and their joint satisfiability $satisfiable_{\mathtt{cs}}((\mathtt{st}_1,\mathtt{c}_1),(\mathtt{st}_2,\mathtt{c}_2))$. To avoid naming conflicts in the traces of the pair, we consider disjoint alphabets $\mathtt{a}_1$ and $\mathtt{a}_2$: those with no common elements.

Example 16. *We list some of the constrained symbolic traces of the process Components in Section 2.1.*

$$(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1 \rangle, \alpha_1 = \beta_1) \tag{3}$$

$$(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1, \mathtt{right}.\gamma_1, \mathtt{left}.\delta_1 \rangle, \alpha_1 = \beta_1 \land \beta_1 = \delta_1) \tag{4}$$

$$(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1, \mathtt{right}.\gamma_1, \mathtt{left}.\delta_1, \mathtt{right}.\epsilon_1 \rangle, \alpha_1 = \beta_1 \land \beta_1 = \delta_1) \tag{5}$$

For the Medium process, some traces are as follows.

$$(\langle \mathtt{left}.\alpha_2 \rangle, \mathtt{true}) \tag{6}$$

$$(\langle \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2 \rangle, \alpha_2 = \beta_2) \tag{7}$$

$$(\langle \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2, \mathtt{left}.\gamma_2 \rangle, \alpha_2 = \beta_2) \tag{8}$$

$$(\langle \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2, \mathtt{left}.\gamma_2 \rangle, 1 - \alpha_2 = \beta_2) \tag{9}$$

$$(\langle \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2, \mathtt{left}.\gamma_2, \mathtt{right}.\delta_2 \rangle, \alpha_2 = \beta_2 \land 1 - \delta_2 = \gamma_2) \tag{10}$$

$$(\langle \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2, \mathtt{left}.\gamma_2, \mathtt{right}.\delta_2 \rangle, 1 - \alpha_2 = \beta_2 \land \delta_2 = \gamma_2) \tag{11}$$

The synchronising pairs for the parallelism between Components and Medium cannot include the empty

trace, or the traces that do not have at least one occurrence of an event representing a communication on left or right. Additionally, for example, the traces (3) and (6) above form a synchronising pair, but (3) and (7) do not, because it is not possible to match the event on right in (7): this pair of traces is not satisfiable. The traces (5) and (10), on the other hand, also form a synchronising pair.

In addition, for illustration, we consider a different action for the Receiver process in Section 2.1 used in the definition of Components. Namely, we consider right?m1 ⟶ right.m1 ⟶ ..., instead of right?m1⟶right?m2⟶.... In this case, once the message m1 is input only another copy of m1 is accepted in the second communication via right: that is, instead of a new input right?m2, we have a synchronisation right.m1 based on m1. So, the constraint of the trace (5) above would be $\alpha_1 = \beta_1 \wedge \beta_1 = \delta_1 \wedge \epsilon_1 = \gamma_1$. Such a trace would not match the trace (10), because matching requires $\epsilon_1 = \delta_2$. This is not consistent with the requirement that $\epsilon_1 = \gamma_1$ because matching also requires $\gamma_1 = 1 - \delta_2$. □

In Definition 12, we require that the projections of the traces are not empty, since otherwise the pair of traces does not really lead to any synchronisation. (In fact, it is enough to require that one of them is not empty, since satisfiability requires them to be of the same length.) The projection $(\mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs})$ of a constrained symbolic trace cst to the set cs is the sequence of channels (rather than events) obtained by keeping from cst only the channels from events representing communications through a channel in cs.

Example 17. *In our example, the projections of (3) and (6) to {| left, right |}, for example, are both ⟨left⟩. The projection of (4) to the same set of channels is ⟨left, right, left⟩. Finally, the projection of (10) is ⟨left, right, left, right⟩.* □

Formal definitions for this operator and others used later are in Appendix A.

Satisfiability for pairs of (synchronising) constrained symbolic traces embeds the fact that the constraints of the traces are jointly satisfiable, even when we match the communications on the synchronisations that they require. This matching gives rise to equalities that strengthen the requirements imposed by the original individual constraints, which must be satisfiable as well.

Definition 13 (Satisfiability). *For every pair $(\mathtt{st}_1, \mathtt{c}_1)$ and $(\mathtt{st}_2, \mathtt{c}_2)$ of constrained symbolic traces and synchronisation channel set cs, we define $satisfiable_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$ as follows.*

$$satisfiable_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2)) \Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_1 \wedge \mathtt{c}_2$$

where $\mathtt{a}_1$ and $\mathtt{a}_2$ are the alphabets of $(\mathtt{st}_1, \mathtt{c}_1)$ and $(\mathtt{st}_2, \mathtt{c}_2)$.

The equalities are defined by the operator $\mathtt{st}_3 =_{\mathsf{S}} \mathtt{st}_4$, which identifies the requirement for synchronisation of the pair of symbolic traces $\mathtt{st}_3$ and $\mathtt{st}_4$ containing only communications that need to synchronise. In the above definition, these traces are $\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}$ and $\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}$. The projection $\mathtt{st} \upharpoonright_{\mathsf{E}} \mathtt{cs}$ is similar to $\mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs}$, but it applies to symbolic traces st and results in symbolic traces: sequences of events, rather than of channels.

Example 18. *The projection of the symbolic trace (4) in Example 16 to the set {| left, right |}, for example, is $\langle \mathtt{left}.\beta_1, \mathtt{right}.\gamma_1, \mathtt{left}.\delta_1 \rangle$. For (10) the result of the projection is this same trace.* □

As formalised below, the result of $\mathtt{st}_1 =_{\mathsf{S}} \mathtt{st}_2$ is a conjunction of equalities that require that the values of the symbolic variables used to represent synchronising communications in $\mathtt{st}_1$ and $\mathtt{st}_2$ are the same.

Definition 14 (Conjunction of symbolic equalities). *We define inductively $\mathtt{st}_1 =_{\mathsf{S}} \mathtt{st}_2$ as follows.*

$$\begin{aligned}
&\langle\rangle =_{\mathsf{S}} \langle\rangle \;=\; \mathtt{true}\\
&\langle\rangle =_{\mathsf{S}} \mathtt{st}_2 \;=\; \mathtt{false}\\
&\mathtt{st}_1 =_{\mathsf{S}} \langle\rangle \;=\; \mathtt{false}\\
&(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \;=\; \alpha = \beta \wedge (\mathtt{st}_1 =_{\mathsf{S}} \mathtt{st}_2)\\
&(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) =_{\mathsf{S}} (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \;=\; \mathtt{false},\ \textit{if}\ \mathtt{d}_1 \neq \mathtt{d}_2
\end{aligned}$$

If $\mathtt{st}_1$ and $\mathtt{st}_2$ have different lengths or require communications over different channels, synchronisation is not possible and the requirement defined by $\mathtt{st}_1 =_{\mathsf{S}} \mathtt{st}_2$ is just false.

Example 19. *For the symbolic traces of (3) and (6), we get the equality* $\beta_1 = \alpha_2$. *For (3) and (7), we get just* `false`. *For (5) and (10), we get the conjunction of equalities* $\beta_1 = \alpha_2 \wedge \gamma_1 = \beta_2 \wedge \delta_1 = \gamma_2 \wedge \epsilon_1 = \delta_2$. □

As seen in Definition 13, in $\mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$, the variables in the alphabets $\mathtt{a}_1$ and $\mathtt{a}_2$ are used in $(\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs})$ and in the constraints $\mathtt{c}_1$ and $\mathtt{c}_2$.

4.2. Synchronisation traces

Using the notion of synchronisation pairs, we define the notion of synchronisation traces, which are constrained symbolic traces that record a parallel execution of a pair of processes (rather than separate executions of the parallel processes captured by synchronisation pairs).

Example 20. *Continuing with our example, from the synchronising pair of traces (3) and (6), we get the synchronising trace* $(\langle \mathtt{in}.\alpha, \mathtt{left}.\beta \rangle, \alpha = \beta)$ *over the alphabet* $\langle \alpha, \beta, \gamma, \ldots \rangle$ *to capture possible observations of the parallel execution of Components and Medium. From (5) and (10), we get the synchronising trace* $(\langle \mathtt{in}.\alpha, \mathtt{left}.\beta, \mathtt{right}.\gamma, \mathtt{left}.\delta, \mathtt{right}.\epsilon \rangle, \alpha = \beta \wedge \beta = \gamma \wedge \beta = \delta \wedge \delta = \mathtt{1} - \epsilon)$. □

As defined below, synchronisation traces are characterised by the sets $\mathit{Scstraces}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$ of constrained symbolic traces, formed from the synchronising constrained symbolic trace pairs in $\mathit{ScstraceP}^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$, which, as formalised in Definition 12, are pairs of traces for the processes $\mathtt{P}_1$ and $\mathtt{P}_2$, using disjoint alphabets $\mathtt{a}_1$ and $\mathtt{a}_2$, and based on the synchronisation set `cs`.

Definition 15 (Synchronising traces). *For every pair of processes* $\mathtt{P}_1$ *and* $\mathtt{P}_2$, *synchronisation channel set* `cs`, *alphabets* $\mathtt{a}_1$, $\mathtt{a}_2$, *and* `a`, *we define the set* $\mathit{Scstraces}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$ *over the alphabet* `a` *as follows.*

$$\mathit{Scstraces}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2) = \bigcup \{\, \mathtt{p} : \mathit{ScstraceP}^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2) \bullet \mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!) \,\}$$

To form traces out of the pairs of traces `p` in $\mathit{ScstraceP}^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$, we apply the synchronisation operator $(\mathtt{st}_1, \mathtt{c}_1) \,[\![\, \mathtt{cs} \,]\!]\, (\mathtt{st}_2, \mathtt{c}_2)$ (Definition 16) to the traces `p`.1 and `p`.2 in `p`. We use $t.i$ to refer to the i-th element of a tuple t. The synchronisation operator identifies all the traces that can arise from the parallel execution of processes that can carry out the traces `p`.1 and `p`.2, synchronising on the channels in `cs`.

Example 21. *In the case of our example, from the synchronising pair of traces (3) and (6), we get the synchronising trace* $(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1 \rangle, \alpha_1 = \beta_1)$. *From (5) and (10), we get a synchronising trace* $(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\alpha_2, \mathtt{right}.\beta_2, \mathtt{left}.\gamma_2, \mathtt{right}.\delta_2 \rangle, \alpha_1 = \alpha_2 \wedge \alpha_2 = \beta_2 \wedge \alpha_2 = \gamma_2 \wedge \gamma_2 = \mathtt{1} - \delta_2)$. □

As illustrated above, the result of the application of `p`.1 [[`cs`]] `p`.2, as formalised in Definition 16, is a set of traces with possibly different alphabets. To obtain a set of traces over the given alphabet `a`, we use in Definition 15 the normalising function $\mathcal{N}^{\mathtt{a}}$ (Definition 18), which basically carries out a renaming.

Synchronisation for constrained symbolic traces is specified in Definition 16 in terms of the similar operator $\mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2$ introduced in Definition 17. This new operator is for symbolic traces $\mathtt{st}_1$ and $\mathtt{st}_2$ and has an extra parameter: a constraint `c`. In the specification of $(\mathtt{st}_1, \mathtt{c}_1) \,[\![\, \mathtt{cs} \,]\!]\, (\mathtt{st}_2, \mathtt{c}_2)$ in Definition 16, it is applied with parameter $\mathtt{c}_1 \wedge \mathtt{c}_2$ to capture the original constraints.

Example 22. *From the synchronising pair of traces (3) and (6), the synchronisation operator for symbolic traces gives the constrained symbolic trace* $(\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1 \rangle, \alpha_1 = \beta_1 \wedge \beta_1 = \alpha_2)$. *Similarly, from (5) and (10), we get a trace with the symbolic trace* $\langle \mathtt{in}.\alpha_1, \mathtt{left}.\beta_1, \mathtt{right}.\gamma_1, \mathtt{left}.\delta_1, \mathtt{right}.\epsilon_1 \rangle$, *and with a constraint that conjoins the original constraints of (5) and (10) and a conjunction of equalities* $\beta_1 = \alpha_2 \wedge \gamma_1 = \beta_2 \wedge \delta_1 = \gamma_2 \wedge \epsilon_1 = \delta_2$ *that forces the matching of their synchronised events.* □

As illustrated above, synchronisation of symbolic traces gives rise to constraints that refer to variables that are not part of the resulting symbolic trace. We, therefore, in defining $(\mathtt{st}_1, \mathtt{c}_1) \,[\![\, \mathtt{cs} \,]\!]\, (\mathtt{st}_2, \mathtt{c}_2)$ below, use the $\mathcal{R}$ function to quantify away the spurious symbolic variables, which are used in the traces of the parallel processes, but are not needed in the synchronisation trace resulting from them.

Definition 16 (Synchronisation operator - constrained symbolic traces). *We define the set of synchronising constrained symbolic traces* $(\mathtt{st}_1, \mathtt{c}_1) \llbracket \mathtt{cs} \rrbracket (\mathtt{st}_2, \mathtt{c}_2)$ *as follows.*

$$(\mathtt{st}_1, \mathtt{c}_1) \llbracket \mathtt{cs} \rrbracket (\mathtt{st}_2, \mathtt{c}_2) \;=\; \mathcal{R}\,(\!|\, \mathtt{st}_1 \llbracket \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \rrbracket \mathtt{st}_2 \,|\!)$$

The synchronisation operator for symbolic traces is simply a symbolic version of the traditional traces operator in [50, p70]. It defines a set of constrained symbolic traces that can arise from synchronisation on the events involving channels in the synchronisation set $\mathtt{cs}$ while performing the traces $\mathtt{st}_1$ and $\mathtt{st}_2$.

Definition 17 (Synchronisation operator - symbolic traces).

$$\mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \;=\; \mathtt{st}_2 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_1$$

$$\langle\rangle \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \langle\rangle \;=\; \{\, (\langle\rangle, \mathtt{c}) \,\} \qquad \textit{[provided } c \textit{ is satisfiable]}$$

$$\langle\rangle \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \langle\rangle \;=\; \varnothing \qquad \textit{[otherwise]}$$

$$\langle\rangle \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \langle \mathtt{d}.\beta \rangle \;=\; \{\, (\langle \mathtt{d}.\beta \rangle, \mathtt{c}) \,\} \qquad \textit{[provided } \mathtt{d} \notin \mathtt{cs}, c \textit{ is satisfiable]}$$

$$\langle\rangle \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \langle \mathtt{d}.\beta \rangle \;=\; \varnothing \qquad \textit{[otherwise]}$$

$$(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \;=\; \{\, \mathtt{cst} : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}$$
[provided $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$*]*

$$(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \;=\; \{\, \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, (\mathtt{c} \wedge \alpha = \beta) \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}$$
[provided $\mathtt{d} \in \mathtt{cs}$*]*

$$(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \;=\; \varnothing \qquad \textit{[provided } \mathtt{d}_1 \neq \mathtt{d}_2, \mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \in \mathtt{cs}\textit{]}$$

$$(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \;= \qquad \textit{[provided } \mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}\textit{]}$$
$$\{\, \mathtt{cst} : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\} \cup$$
$$\{\, \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \bullet (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}$$

We, first of all, establish that the synchronisation operator is commutative, and then provide an inductive definition. If both traces are empty, then we get only the empty trace itself, with constraint $\mathtt{c}$. A proviso guarantees that $\mathtt{c}$ is satisfiable, otherwise no synchronisation traces arise. If there is a conflict in the synchronisation requirements of the traces, then no synchronisation arises either. A conflict occurs when one of the traces requires synchronisation on an event on which the other is not ready to agree (either because it is empty, or its next event is a communication on a different channel also in the synchronisation set.) When there is a synchronisation on events $\mathtt{d}.\alpha$ and $\mathtt{d}.\beta$, we make a(n arbitrary) choice to use the symbolic variable β in the resulting synchronisation trace. This means that α is not used in that trace, but is mentioned in the constraint $\mathtt{c} \wedge \alpha = \beta$. To remove any such symbolic variables that become redundant, in Definition 16, we apply $\mathcal{R}$ to each of the traces in $(\mathtt{st}_1, \mathtt{c}_1) \llbracket \mathtt{cs} \rrbracket (\mathtt{st}_2, \mathtt{c}_2)$.

As already said, the traces in $\mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2$ are (potentially) over different alphabets. In the definition of $Scstraces_{\mathtt{cs}}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1, \mathtt{P}_2)$ above, the normalisation function $\mathcal{N}$ ensures that the constrained symbolic traces resulting from the synchronisation are over the alphabet $\mathtt{a}$. Alphabets keep the definition of trace equivalence simple. When defining compositions of (local) traces, however, we have the added complication of requiring normalisation. The normalisation function $\mathcal{N}$, which is defined below, however, is just a renaming.

Definition 18 (Normalisation). *For every* $(\mathtt{st}, \mathtt{c})$ *and alphabet* $\mathtt{a}$*, we define* $\mathcal{N}^{\mathtt{a}}(\mathtt{st}, \mathtt{c})$ *as follows.*

$$\mathcal{N}^{\mathtt{a}}(\mathtt{st}, \mathtt{c}) = (\mathtt{st}[\mathtt{a}/\alpha\mathtt{st}], \mathtt{c}[\mathtt{a}/\alpha\mathtt{st}])$$

This just changes the alphabet of $(\mathtt{st}, \mathtt{c})$, whatever that is, to $\mathtt{a}$.

The choice of alphabets $\mathtt{a}_1$ and $\mathtt{a}_2$ to express the traces of the parallel processes $\mathtt{P}_1$ and $\mathtt{P}_2$ is irrelevant when calculating $Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$ (see Lemma 1 in Appendix C). The alphabet of the traces in this set is always $\mathtt{a}$. From now on, therefore, we omit these alphabets.

Our main result in this section is given by the following theorem, which establishes that every synchronising trace in $Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$ is a constrained symbolic trace of the parallelism $\mathtt{P}_1 \llbracket \mathtt{cs} \rrbracket \mathtt{P}_2$. In addition, they identify all traces of the parallelism that include at least one synchronisation.

Theorem 2.

$$Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2) = \{\, \mathtt{cst} : cstraces^{\mathtt{a}}(\mathtt{P}_1 \llbracket \mathtt{cs} \rrbracket \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \,\}$$

This theorem establishes that if we build tests from $Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$, then we get a subset of the exhaustive test set. Its proof, and those of other theorems introduced here, can be found in Appendix D.

4.3. Compositionality of parallelism

So far, we have, for the sake of clarity and motivation, considered only a parallelism of two processes $\mathtt{P}_1$ and $\mathtt{P}_2$. It has been observed [35] that in practice the manifestation of a large majority of concurrency bugs involves no more than two threads (which correspond to parallel processes in *Circus*). However, we want to consider arbitrary process expressions; for instance, if we have a parallelism $(\mathtt{P}_1 \llbracket \mathtt{csA} \rrbracket \mathtt{P}_2) \llbracket \mathtt{csB} \rrbracket \mathtt{P}_3$, then we may be interested in the synchronisations required by both $\mathtt{csA}$ and $\mathtt{csB}$. In this case, one of the parallel components is not a simple process, but a parallelism $(\mathtt{P}_1 \llbracket \mathtt{csA} \rrbracket \mathtt{P}_2)$. A compositional approach, in which we build tests based on the (local) traces of $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$, would be ideal.

To discuss this approach, we consider generalisations of the definitions of $ScstraceP^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$ and $Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$ where the sets of trace pairs or traces are built, not from processes, but from sets of traces $\mathtt{csts}_1$ and $\mathtt{csts}_2$. The definitions of $ScstraceP_{\mathtt{cs}}(\mathtt{csts}_1,\mathtt{csts}_2)$ and $Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\mathtt{csts}_1,\mathtt{csts}_2)$ are very similar to those of $ScstraceP^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$ and $Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$, and are in Appendix A.

The problem that we face is that selection of synchronisation traces is actually not compositional. Precisely, if we consider $(\mathtt{P}_1 \llbracket \mathtt{csA} \rrbracket \mathtt{P}_2) \llbracket \mathtt{csB} \rrbracket \mathtt{P}_3$ as suggested above, for instance,

$$\begin{array}{l} Scstraces^{\mathtt{a}}_{\mathtt{csB}}(Scstraces^{\mathtt{a}_{12}}_{\mathtt{csA}}(cstraces^{\mathtt{a}_1}(\mathtt{P}_1), cstraces^{\mathtt{a}_2}(\mathtt{P}_2)), cstraces^{\mathtt{a}_3}(\mathtt{P}_3)) \\ \neq \\ Scstraces^{\mathtt{a}}_{\mathtt{csB}}(cstraces^{\mathtt{a}_{12}}(\mathtt{P}_1 \llbracket \mathtt{csA} \rrbracket \mathtt{P}_2), cstraces^{\mathtt{a}_3}(\mathtt{P}_3)) \end{array}$$

An example illustrates the issue here.

Example 23. *The (stateless) processes that we consider are as follows.*

$$\begin{array}{l} P_1 = a \longrightarrow \mathbf{Stop} \,\Box\, b \longrightarrow \mathbf{Stop} \\ P_2 = a \longrightarrow \mathbf{Stop} \\ P_3 = b \longrightarrow \mathbf{Stop} \end{array}$$

We then consider the specification $P \mathrel{\widehat=} (P_1 \llbracket \{\!| a |\!\} \rrbracket P_2) \llbracket \{\!| b |\!\} \rrbracket P_3$. *First of all, we consider the component* $P_1 \llbracket \{\!| a |\!\} \rrbracket P_2$, *which is itself a parallelism. The traces of this parallelism are* $\langle\,\rangle$, $\langle \mathtt{a} \rangle$, *and* $\langle \mathtt{b} \rangle$. *Its only synchronisation trace, on the other hand, is* $\langle \mathtt{a} \rangle$. *If we use this synchronisation trace as a basis to construct the synchronisation traces of* P, *we get the empty set of traces. It, however, has synchronisation trace* $\langle \mathtt{b} \rangle$, *and if we want to cover both synchronisations of the specification, we actually need* $\langle \mathtt{a} \rangle$ *and* $\langle \mathtt{b} \rangle$. □

This means that, in general, to construct a test set to cover all synchronisations in a model, we have to construct a test set for each of the parallelisms, considering the whole set of constrained symbolic traces of each of the components, and not only their synchronisation traces. If the synchronisation sets of the parallelisms are the same, however, we do have compositionality, as established by the following theorem.

Theorem 3.

$$Scstraces^{\mathtt{a}}_{\mathtt{cs}}(Scstraces^{\mathtt{a}_{12}}_{\mathtt{cs}}(cstraces^{\mathtt{a}_1}(\mathtt{P}_1), cstraces^{\mathtt{a}_2}(\mathtt{P}_2)), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$$
$$=$$
$$Scstraces^{\mathtt{a}}_{\mathtt{cs}}(cstraces^{\mathtt{a}_{12}}(\mathtt{P}_1 \llbracket\ \mathtt{cs}\ \rrbracket \mathtt{P}_2), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$$

To consider arbitrary process expressions, however, this is not enough.

In general, we need to understand how to construct tests to cover a synchronisation between parallel components occurring in any part of a system model. This is addressed in the next section.

4.4. Compositionality of other operators

We define below the function $selectS^{\mathtt{a}}(\mathtt{P})$, which includes the set of all constrained symbolic traces of P over the alphabet a that cover at least one synchronisation in P, if any. The set of tests defined from all traces in $selectS^{\mathtt{a}}(\mathtt{P})$ covers all reachable synchronisations of P in all possible ways.

Definition 19 (Selection).

$$selectS^{\mathtt{a}}(\mathtt{begin\ state}\ [\mathtt{x} : \mathtt{T}] \bullet \mathtt{A}\ \mathtt{end}) = \varnothing$$

$$selectS^{\mathtt{a}}(\mathtt{P}_1 \llbracket\ \mathtt{cs}\ \rrbracket \mathtt{P}_2) = Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$$

$$selectS^{\mathtt{a}}(\mathtt{P}_1 \,\square\, \mathtt{P}_2) = selectS^{\mathtt{a}}(\mathtt{P}_1) \cup selectS^{\mathtt{a}}(\mathtt{P}_2)$$

$$selectS^{\mathtt{a}}(\mathtt{P}_1 \sqcap \mathtt{P}_2) = selectS^{\mathtt{a}}(\mathtt{P}_1) \cup selectS^{\mathtt{a}}(\mathtt{P}_2)$$

$$selectS^{\mathtt{a}}(\mathtt{P}_1\ ;\ \mathtt{P}_2) = selectS^{\mathtt{a}}(\mathtt{P}_1) \cup \mathcal{N}^{\mathtt{a}}(\!|\, tcstraces^{\mathtt{a}_1}(\mathtt{P}_1)\ \text{ccat}\ selectS^{\mathtt{a}_2}(\mathtt{P}_2) \,|\!) \qquad \textit{[provided}\ \text{disjoint}(\mathtt{a}_1, \mathtt{a}_2)\textit{]}$$

$$selectS^{\mathtt{a}}(\mathtt{P} \setminus \mathtt{cs}) = \mathcal{N}^{\mathtt{a}}(\!|\, (\!|selectS^{\mathtt{a}}(\mathtt{P})|\!) \upharpoonright_{\mathtt{C}} \mathtt{cs} \,|\!)$$

A basic process, defined by specifying its state and action, rather than in terms of other processes, has no synchronisation traces. The synchronisation traces of a parallelism $\mathtt{P}_1 \llbracket \mathtt{cs} \rrbracket \mathtt{P}_2$ are given by $Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\mathtt{P}_1, \mathtt{P}_2)$. As established by Theorem 3, in selecting traces for a parallelism, we can take advantage of nested parallelisms over the same synchronisation set to optimise the procedure and proceed in a compositional way. This is not considered in the specification of $selectS^{\mathtt{a}}(P)$, but can be exploited in an algorithm.

The synchronisation traces of a sequence $\mathtt{P}_1\ ;\ \mathtt{P}_2$ include those of $\mathtt{P}_1$ and the continuations of the terminating traces of $\mathtt{P}_1$ that cover the synchronisations of $\mathtt{P}_2$. To define these, we use the set $tcstraces^{\mathtt{a}}(\mathtt{P})$ of terminating traces of a process P. Its definition below is similar to that of $cstraces^{\mathtt{a}}(\mathtt{P})$; a terminating trace is identified by its leading to a configuration $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip})$ for **Skip**.

Definition 20 (Terminating traces).

$$tcstraces^{\mathtt{a}}(\mathtt{begin\ state}[\mathtt{x} : \mathtt{T}] \bullet \mathtt{A}\ \mathtt{end}) = tcstraces^{\mathtt{a}}(\mathtt{w}_0 \in \mathtt{T}, \mathtt{x} := \mathtt{w}_0, \mathtt{A})$$

$$tcstraces^{\mathtt{a}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1) = \{\mathtt{st}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\}$$

To cover the synchronisations in $\mathtt{P}_2$, we need to prefix its synchronisation traces with terminating traces of $\mathtt{P}_1$. We use the operator $\mathtt{csts}_1$ ccat $\mathtt{csts}_2$; the traces in $\mathtt{csts}_1$ ccat $\mathtt{csts}_2$ are those formed by concatenating all pairs of symbolic traces and conjoining their associated constraints. Since, in general, the length of the traces in $tcstraces^{\mathtt{a}_1}(\mathtt{P}_1)$ vary, it is not possible to predict the alphabet of the traces in $tcstraces^{\mathtt{a}_1}(\mathtt{P}_1)$ ccat $selectS^{\mathtt{a}_2}(\mathtt{P}_2)$. In general, different traces in this set may have different alphabets. We, therefore, use in Definition 19 the normalisation function $\mathcal{N}$ to obtain a set of traces over the alphabet a.

For a hiding $\mathtt{P} \setminus \mathtt{cs}$, since we are interested in covering the hidden synchronisations, we consider the synchronisation traces of P. We remove from them, however, the events that represent communications through the channels in cs. For that, we use the projection function $\mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs}$, whose resulting constrained symbolic trace contains only the events of cst that are not communications over channels in cs. The alphabet of a symbolic trace resulting from the application of this operator is not predictable, so we again use the normalisation function to obtain in $\mathtt{selectS}^{\mathtt{a}}(\mathtt{P} \setminus \mathtt{cs})$ a set of constrained symbolic traces over a.

Example 24. *Both synchronising traces in Example 20 give rise to the synchronising trace* $(\langle \mathtt{in}.\alpha \rangle, \mathtt{true})$ *of Protocol.* □

Our main result here is introduced below: our selection strategy identifies a subset of the traces of a process.

Theorem 4.

$$selectS^{\mathtt{a}}(\mathtt{P}) \subseteq cstraces^{\mathtt{a}}(\mathtt{P})$$

This means that selection based on $selectS^{\mathtt{a}}(\mathtt{P})$ identifies an unbiased test set (that is, a set of traces that can be used to construct an unbiased test set). In terms of the framework defined in the previous section, the property $\pi(t, P)$ that can be used to achieve selection based on synchronisation coverage is $t \in selectS^{\mathtt{a}}(\mathtt{P})$. We consider the use of $selectS^{\mathtt{a}}(\mathtt{P})$ for selection of tests for traces refinement next.

4.5. More selective synchronisation coverage

With the property $\pi(t, SP) = t \in selectS^{\mathtt{a}}(SP)$, we can define, using our framework from Section 3, two exhaustive test sets: one for existential and one for universal selection based on traces. In this example, the traces and tests are symbolic. We use the framework in Section 3, but with definitions adapted (in the expected and straightforward way) to consider symbolic traces and tests. The symbolic exhaustive test sets are shown below. In these definitions, we also rely on Theorem 4 to conclude that $t \in selectS^{\mathtt{a}}(SP)$ implies that $t \in cstraces(SP)$ and simplify the definitions obtained directly using (1) and (2).

$$SyncExist_T^{trace}(SP) = \{AllTests_T(cst, SP) \mid cst \in selectS^{\mathtt{a}}(SP)\}$$
$$SyncUniv_T^{trace}(SP) = \{ST_T(cst, cse) \mid cst \in selectS^{\mathtt{a}}(SP) \land cse \in \overline{csinitials}(SP, cst)\}$$

Existential selection, in this example, is not very interesting, since its goal is to run only one test covering at least one synchronisation. Universal selection, on the other hand, is the basis for the property: "all tests that involve at least one synchronisation in the trace are attempted".

The problem that we face in this section is the fact that the above sets of symbolic tests are still infinite, and, in addition, each symbolic test has an infinite number of instantiations as well. We consider next how to obtain a finite set of symbolic tests for universal selection.

For selection of symbolic tests, what we suggest is the use of a regularity hypothesis. If the model does not have unbounded nondeterminism, a regularity hypothesis that limits the size of the symbolic traces defines a finite set of symbolic tests. It is well known, however, that the choice of the limiting size must be careful, to avoid throwing away too many interesting tests [18]. In the case of synchronisation coverage, it is sensible to limit the size on the traces in such a way that ensures that all channels are used at least once.

Example 25. *For synchronisation tests for the parallelism between the processes Components and Medium, the bound should be at least 8, to ensure that traces with events representing communications on out are included. For this system, we define the symbolic exhaustive set of tests (with respect to synchronisation coverage and our regularity hypothesis), as follows.*

$$FSyncUniv_T^{trace}(Components \,[\![\, \{\!|\, left, right \,|\!\} \,]\!]\, Medium) =$$
$$\{\, ST \mid ST \in SyncUniv_T^{trace}(Components \,[\![\, \{\!|\, left, right \,|\!\} \,]\!]\, Medium) \land \#\, ST \leq 9 \,\}$$

We use the operator $\#\, ST$ *to denote the size of a symbolic test: the size of its symbolic trace plus one.* □

For selection of instances of symbolic tests, what we suggest is the use of uniformity hypotheses based on the constraint of the symbolic traces and of the symbolic forbidden continuations. We can use the constraints to take into account information about the updates of data specified in a symbolic trace; an extreme decomposition uses a DNF characterisation of the constraints to define several domains. To be more selective, we consider each constraint as defining a single domain. In this case, selection is based on the interactions and paths of execution in the model defined by the operational semantics used to generate the symbolic traces. This means that we take the existential selection of instances of the symbolic tests (discussed in Section 3.4). This runs, if possible, one instance of every selected symbolic test.

Example 26. *The driver for existential selection of instances of the tests in Example 25 is as follows.*

$$SyncUnivExistDriver^{trace}(Components \llbracket \{\!| left, right |\!\} \rrbracket Medium, SUT) =$$
$$\Big|\Big|\Big| ST : FSyncUniv_T^{trace}(Components \llbracket \{\!| left, right |\!\} \rrbracket Medium) \bullet$$
$$(\Big|\Big|\Big| T : instTest(ST) \bullet Execution_{SUT}^{SP}(T)) \llbracket \{\!| inc, pass, fail |\!\} \rrbracket TMonitor$$

For each of the finitely many symbolic tests in $FSyncUniv_T^{trace}(Components \llbracket \{\!| left, right |\!\} \rrbracket Medium$*, we have the independent execution of its instances under the control of a TMonitor. As soon as an instance reaches a conclusive verdict, other instances are discarded.* □

We observe that, as illustrated in the above example, the search for a single instance may still not terminate, if none is implemented. Static choice of an instance is not a good idea, as extensively discussed in Section 3.

The exhaustivity of our test sets rely on the complete test coverage assumption to guarantee, when the SUT is nondeterministic, the execution of all its behaviours. In the case of parallel composition, it implies the spontaneous coverage of all synchronisation-race variants: this assumption can be weakened by using specific schedulers when driving the tests, such as in reachability testing [34], or CHESS [1].

5. Related works

One of the first well known coverage criteria for concurrent programs was developed by Taylor et al. [55] for Ada. It is based on notions of concurrency states and concurrency graphs used to define classical-graph coverage criteria (all-concurrency-paths, all-edges-between-cc-states, and all-cc-states), and a weaker criterion called *all-possible-rendezvous*, which requires that for all concurrency states that involve a rendezvous, there is at least one path through this state. This notion of coverage hides the sequential activities in the concurrency states, thus focusing on synchronisation anomalies. The authors also sketch the idea of forcing particular concurrency paths when driving test execution. As mentioned in Section 4, synchronisation coverage for Ada programs has been more precisely defined and implemented by Bron et al. [5].

Carver, Lei, Tai and others has developed reachability testing [7, 34], which provides ways to execute deterministically a concurrent program and reproduce a given test case. It is a purely dynamic method for generating *synchronisation sequences* (SYN-sequences) on the fly. Reachability testing algorithms are exhaustive; this ensures the execution of all the possible SYN-sequences for a given input, provided there is no unbounded loop. In this case, a bound on the length of the SYN-sequences is introduced.

For synchronous message passing, like in *Circus*, SYN-sequences contain pairs of sending and receiving events exercised in an execution. Deterministic execution is led according to a given SYN-sequence. Reachability testing does not address selection of test inputs or SYN-sequences, but specification-based reachability testing is addressed in [8] with a limited specification language for expressing local constraints on the SYN-sequences. When the only source of nondeterminism is race variants, reachability testing can be used to ensure the complete test assumption used in our work, since it ensures execution of all SYN-sequences.

The use of a reachability graph for structural test-sequence generation for concurrent programs is reported in [61]. A more recent approach by Souza et al. [54] reports some experiments with reachability testing and a few coverage criteria based on a notion of *sync-edge association*, which seems similar to the coverage of those edges corresponding to a synchronisation pair. This is close to the notion of synchronisation coverage that we proposed for *Circus*, but it is based on static analysis of the program.

The next generation of reachability testing algorithms is embedded in CHESS [39, 1]. This tool repeatedly runs concurrent tests ensuring that every run takes a different interleaving. If an interleaving results in an error, CHESS can reproduce it for improved debugging. The only perturbation introduced is a thin wrapper layer between the SUT and the concurrency API. The wrappers are similar to implementations of our drivers, but just like in all reachability testing techniques, they are not concerned with selection.

Dynamic test generation [6] avoids infeasible paths when generating tests from the text of a program and covering all paths. This is achieved by symbolic and standard execution in an integrated way. Test generation starts by running the program on an arbitrary concrete test input, and recording the choices and

the path followed. Afterwards, other tests are generated by switching choices to execute another path: a constraint solver is used to check that the changes lead to feasible paths. This method has been mainly applied to testing based on the source code of programs, as in PathCrawler [60], CUTE [53], and Dart [25].

A first difference between these approaches and our work is that our models are potentially more abstract than the source code of programs. Secondly, similarity comes from the need to attempt alternative paths when a test is not possible, but the main difference is that the SUT is not required to implement all the traces of our models. Thus, our concern is absence of specification traces rather than unsatisfiability, so we do not use constraint solving. We note that in the presence of loops, dynamic test generation may not terminate. Termination is ensured by introducing some limit to the length of paths, which corresponds to an implicit regularity hypothesis on the SUT. Similarly, our drivers for symbolic tests may not terminate.

Adaptive testing has been studied for the derivation of elaborate test experiments from nondeterministic finite state machines [33] for possibly nondeterministic implementations [46, 27, 29]. The idea is to take into account the output of the SUT to determine how to proceed with the experiment. It means that there is no preset input sequence, but a rooted tree that anticipates all possible sequences. It is very similar to the factorised tree-shaped tests we present in Section 3.3, which lets the SUT to lead the test. The role of our various test drivers is a generalisation of this approach, where, in addition to the possibility of non-implemented traces, we introduce a variety of selection strategies.

For a deterministic SUT, further work [30] has considered also the optimisation of a testing experiment by ordering the adaptive tests. In our setting, we can consider, for example, that, if the SUT is deterministic, an inconclusive test verdict indicates that all tests for the same trace or any of its extensions is also going to produce an inconclusive verdict. Therefore, they are useless and can be eliminated. Sophisticated drivers are required to order the tests and perform such optimisations.

An important feature that is missing in the semantics of *Circus* (and CSP) is a distinction of inputs and outputs. The difference in the controllability of these events, with inputs controlled by the environment and outputs by the process, is clearly important for testing. The work in [16] discusses this point in the context of CSP and, in particular, the impact of a refinement relation that captures the distinction between inputs and outputs on its testing theory. We note, however, that the distinction between inputs and outputs affects the refusals of a process, but not its traces. For this reason, the results presented here for traces refinement are not affected by considerations of controllability.

Peleska and Siegel [44, 45] applied CSP-based testing and developed a tool and technique for automation. More recently, CSP was used by Nogueira et al. to formalise a notion of conformance traditionally associated with input-output labelled transition systems [41], with refinement model checking used for generation and selection of tests. This work provides a technique and a tool for generation and selection of tests based on the use of FDR, the CSP refinement model checker. In both these works, universal selection strategies are used, and, therefore, the issue on non-implemented traces does not arise.

In [37] Massink et al. have studied testing based on statecharts. A testing theory, in the sense of [40], that is, for models rather than systems, as well as a notion of conformance and a test-case generation algorithm, were presented. The theoretical framework developed follows a pattern very similar to ours, but the considered language and the conformance relation are very different. Exhaustiveness of the generated test set is proved, but test selection strategies are left as future work.

A symbolic version of the *ioco* relation [57] was stated by Frantzen et al. in [22] for Input-Output Symbolic Transition Systems (IOSTS) implemented by input-enabled IOSTS. There is a slightly different notion of constrained symbolic extended traces, with two constraints, one on the interaction variables, and another on the updates of the state variables. Location coverage is mentioned as a perspective: a sort of symbolic state coverage. This would raise the same issues we have studied and solved here, since it is an existential coverage criterion, and *ioco* does not require all the traces to be implemented.

6. Conclusions

In this paper, we have identified and illustrated the problem of selection of tests for establishing traces refinement. In short, it is not possible to have a static selection, because a correct SUT may not implement the selected traces, in which case coverage ensured statically may not be achieved during experiments.

As a solution, we have proposed the use of strategies that combine the definition of an exhaustive test set with respect to the selection criteria of interest and a driver. Although, we have considered the main kinds of selection criteria, it is not ever possible to claim that we have catered for every imaginable criterion. We believe, however, that the approaches that we have proposed and illustrated for comprehensive categories of criteria can be usefully combined or extended to cater for sophisticated selection criteria. We have, in this paper, considered selection of both concrete and symbolic tests.

We have considered criteria based on one or several (multiple) properties of traces or of tests, and that require one (existential) or all (universal) implemented traces or tests that satisfy the property(ies) to be run. For each such kind of criteria, we have formalised the construction of the exhaustive test set and described a driver using *Circus* constructs. These driver specifications provide an accurate description of the requirements of the test experiments, but practical techniques have to consider efficient use of the SUT and of constraint solvers to avoid the use of too many copies of the SUT or instantiations of symbolic tests.

To illustrate the use of our framework, we have formalised a selection criterion for testing based on parallel *Circus* specifications. It is appropriate for integration testing, and requires coverage of synchronisations between parallel processes in a specification. We have formally defined this trace selection criterion and, specified an algorithm for selection. We have proved that this algorithm generates traces of the specification, and, therefore, unbiased tests. This is used to define a finite set of symbolic tests and accompanying driver.

The problem raised by the possibility of selecting non-implemented traces is also raised by a different conformance relation that we have studied previously in the context of *Circus*, namely, *conf*. What *conf* requires is that, for traces that are common to the specification and the SUT, the SUT deadlocks only when a deadlock is allowed in the model. As opposed to the tests for traces refinement, tests for *conf* are positive. They check that, after a given trace of the specification, at least one event from a set of minimal acceptances is accepted by the SUT. Since *conf* is restricted to traces that are common to the SUT and the specification, if the trace is not implemented by the SUT, we again have an inconclusive verdict. We show in [14] that the same solutions proposed here are applicable in testing for *conf*.

This is an important result because we have previously shown that failures refinement, another important conformance relation for both CSP and *Circus*, can be characterised as the conjunction of traces refinement and *conf* [10, 13]. An exhaustive test set for failures refinement is, therefore, the union of the exhaustive test sets for traces refinement and for *conf*. Moreover, while traces refinement is concerned with safety properties and cannot, for instance, detect a mismatch in the alphabet of events of an SUT, *conf* can be used to identify such a problem. If the SUT insists in performing an event outside of the alphabet of the specification, unless a deadlock is allowed by the specification, a test for *conf* can identify the problem.

A popular conformance relation is *ioco* [57]. Since it allows that some traces of the specification not to be implemented, our results are of interest for test selection in the case of *ioco*.

As for future work we are interested in performing case studies, both in the form of the definition of additional selection criteria and experiments of testing actual systems. In that respect, the automation of the *Circus* testing theory presented in [19, 20] is of great relevance.

Also of interest is the application of the ideas presented here to the notion of traces for *Circus* defined in [12] and [15]. In that work, we consider traces that record information about the data operations (specification statements, assignments, guards, and so on) in the *Circus* specification, and support selection techniques based on the structure of the model: in [15] we present several data flow coverage criteria for *Circus* specifications. Just like the symbolic traces of the original *Circus* testing theory, these symbolic traces do not need to be implemented by an SUT. So, the same issues raised here apply.

Acknowledgements. We are grateful to the Royal Society and the CNRS for funding our collaboration via their International Exchange scheme, and to Rob Hierons for various discussions. We are also indebted to anonymous referees for their careful revision and useful suggestions.

References

[1] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 420 – 434. Springer, 2010.

[2] G. Barrett. Model checking in practice: The T9000 Virtual Channel Processor. *IEEE Transactions on Software Engineering*, 21(2):69 – 78, 1995.

[3] L. Bentakouk, P. Poizat, and F. Zaidi. Checking the behavioral conformance of web services with symbolic testing and an smt solver. In Springer, editor, *Tests and Proofs*, volume 6706 of *Lecture Notes in Computer Science*, pages 33 – 50, 2011.

[4] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387 – 405, 1991.

[5] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206 – 212, 2005.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *33rd International Conference on Software Engineering*, pages 1066 – 1071. ACM, 2011.

[7] R. H. Carver and K.-C. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66 – 74, 1991.

[8] R. H. Carver and Kuo-Chung Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 24:471 – 490, 1998.

[9] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From Control Law Diagrams to Ada via *Circus*. *Formal Aspects of Computing*, 23(4):465 – 512, 2011.

[10] A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods*, volume 4789 of *Lecture Notes in Computer Science*, pages 151 – 170. Springer-Verlag, 2007.

[11] A. L. C. Cavalcanti and M.-C. Gaudel. A note on traces refinement and the *conf* relation in the Unifying Theories of Programming. In A. Butterfield, editor, *Unifying Theories of Programming 2008*, volume 5713 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[12] A. L. C. Cavalcanti and M.-C. Gaudel. Specification Coverage for Testing in *Circus*. In S. Qin, editor, *Unifying Theories of Programming*, volume 6445 of *Lecture Notes in Computer Science*, pages 1 – 45. Springer-Verlag, 2010.

[13] A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in *Circus*. *Acta Informatica*, 48(2):97 – 147, 2011.

[14] A. L. C. Cavalcanti and M.-C. Gaudel. A note on test selection for conf refinement. Rapport LRI 1569, Université de Paris-Sud, 2013. Available at `www.lri.fr/ bibli/Rapports-internes/2013/RR1569.pdf`.

[15] A. L. C. Cavalcanti and M.-C. Gaudel. Data Flow coverage for *Circus*-based testing. In *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, 2014.

[16] A. L. C. Cavalcanti and R. M. Hierons. Testing with Inputs and Outputs in CSP. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 359 – 374, 2013.

[17] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 – 181, 2003.

[18] R. K. Dong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):103 – 130, 1994.

[19] A. Feliachi. *Semantics-Based Testing for Circus*. PhD thesis, LRI, Université de Paris-Sud, 2012.

[20] A. Feliachi, M. C. Gaudel, M. Wenzel, and B. Wolff. The *Circus* Testing Theory Revisited in Isabelle/HOL. In L. Groves and J. Sung, editors, *15th International Conference on Formal Engineering Methods*, volume 8144 of *Lecture Notes in Computer Science*, pages 243 – 260. Springer, 2013.

[21] C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *The Z Formal Specification Notation*. Springer-Verlag, 1998.

[22] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, number 4262 in Lecture Notes in Computer Science, pages 40 – 54. Springer, 2006.

[23] L. Freitas and J. P. McDermott. Formal methods for security in the xenon hypervisor. *International Journal on Software Tools for Technology Transfer,*, 13(5):463 – 489, 2011.

[24] S. Fujiwara and G. Bochmann. Testing non-deterministic finite state machines with fault coverage. In *4th International Workshop on Protocol Test Systems*, pages 267–280. North-Holland Publishing Co., 1991.

[25] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, PLDI*, pages 213 – 223. ACM, 2005.

[26] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18 – 25, 2002.

[27] R. M. Hierons. Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. *Computer Journal*, 41(5):349 – 355, 1998.

[28] R M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Transactions on Software Engineering and Methodology*, 11(4):427–448, 2002.

[29] R. M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Information Processing Letters*, 98(2):56 – 60, 2006.

[30] R. M. Hierons and H. Ural. Reducing the cost of applying adaptive test cases. *Computer Networks*, 51(1):224 – 238, 2007.

[31] E. Itoh, Y. Kawaguchi, Z. Furukawa, and K. Ushijima. Ordered Sequence Testing Criteria for Concurrent Programs and the Support Tool. In *1st Asia-Pacific Software Engineering Conference*, pages 236 – 245. IEEE, 1994.

[32] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *5th ACM International Conference on Embedded Software*, pages 299 – 306. ACM, 2005.

[33] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090 – 1126, 1996.

[34] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineeding*, 32(6):382 – 403, 2006.

[35] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329 – 339, 2008.

[36] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95 – 104. IEEE Computer Society Press, 1998.

[37] M. Massink, D. Latella, and S. Gnesi. On testing UML statecharts. *The Journal of Logic and Algebraic Programming*, 69(1-2):1 – 74, 2006.

[38] A. Miyazawa and A. L. C. Cavalcanti. Refinement-oriented models of Stateflow charts. *Science of Computer Programming*, 77(10 – 11):1151 – 1177, 2012.

[39] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Arumuga Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267 – 280. USENIX Association, 2008.

[40] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 3(1-2):83 – 133, 1984.

[41] S. Nogueira, A. C. A. Sampaio, and A. C. Mota. Guided Test Generation from CSP Models. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigün, editors, *5th International Colloquium on Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 258 – 273. Springer, 2008.

[42] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs Using Circus*. PhD thesis, University of York, 2006.

[43] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3 – 32, 2009.

[44] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. In *Formal Methods Europe, Industrial Benefits and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, 1996.

[45] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53 – 77, 1997.

[46] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In *6th International Workshop on Protocol Test systems VI*, pages 363 – 378. North-Holland Publishing Co., 1994.

[47] C. Robinson-Mallett, R. M. Hierons, and P. Liggesmeyer. Achieving communication coverage in testing. *ACM SIGSOFT Software Engineerig Notes*, 31(6):1 – 10, 2006.

[48] C. Robinson-Mallett, R. M. Hierons, J. Poore, and P. Liggesmeyer. Using communication coverage criteria and partial model generation to assist software integration testing. *Software Quality Control*, 16(2):185 – 211, 2008.

[49] M. Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42 – 71, 2006.

[50] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[51] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[52] S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J. Bowen, M. Henson, and K. Robinson, editors, *ZB'2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 416 – 435, 2002.

[53] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263 – 272. ACM, 2005.

[54] S. R. S. Souza, P. S. L. Souza, M. C. C. Machado, M. S. Camillo, A. S. Simão, and E. Zaluska. Using Coverage and Reachability Testing to Improve Concurrent Program Testing Quality. In *23rd International Conference on Software Engineering and Knowledge Engineering*, pages 207 – 212, 2011.

[55] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, pages 206 – 215, 1992.

[56] N. Tillmann, J. de Halleux, and W. Schulte. Parameterized Unit Testing with Pex: Tutorial. In P. H. M. Borba, A. L.C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock, editors, *Testing Techniques in Software Engineering, Second Pernambuco Summer School on Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 141 – 202. Springer, 2010.

[57] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127 – 146. Springer-Verlag, 1996.

[58] J. Tretmans and E. Brinksma. TorX: Automated Model Based Testing. In *1st European Conference on Model-Driven Software Engineering*, pages 13 – 25, 2003.

[59] M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-the-fly testing of reactive systems. Technical Report MSR-TR-2005-03, Microsoft Research, 2005.

[60] N. Williams, B. Marre, P. Mouy, and M. Roger. PathCrawler: Automatic generation of path tests by combining static and dynamic analysis. In *5th European Dependable Computing Conference*, volume 3463 of *Lecture Notes in Computer Science*, pages 281 – 292. Springer, 2005.

[61] W. E. Wong and Y. Lei. Reachability graph-based test sequence generation for concurrent programs. *International Journal of Software Engineering and Knowledge Engineering*, 18(6):803 – 822, 2008.

[62] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

Appendix A. Formal definitions of operators

Definition 21. *We define a pair of alphabets* $\mathtt{a}_1$ *and* $\mathtt{a}_2$ *to be disjoint if their sets of elements are disjoint:* $\mathrm{disjoint}(\mathtt{a}_1, \mathtt{a}_2) = \mathrm{ran}\,\mathtt{a}_1 \cap \mathrm{ran}\,\mathtt{a}_2 = \varnothing$.

Definition 22. *For every constrained symbolic trace* $(\mathtt{st}, \mathtt{c})$ *and synchronisation channel set* $\mathtt{cs}$*, we define the projection* $(\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs}$ *of the trace* $(\mathtt{st}, \mathtt{c})$ *to channels in* $\mathtt{cs}$ *as* $(\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs} = \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs}$*, where the similar projection operator for symbolic traces is defined inductively as follows.*

$$
\begin{array}{lcl}
\langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \langle\rangle \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \langle \mathtt{d} \rangle \frown (\mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs}),\ \textit{if}\ \mathtt{d} \in \mathtt{cs} \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs},\ \textit{if}\ \mathtt{d} \notin \mathtt{cs}
\end{array}
$$

Definition 23. *For every symbolic trace* $\mathtt{st}$ *and synchronisation channel set* $\mathtt{cs}$*, we define inductively the projection* $\mathtt{st} \upharpoonright_{\mathsf{E}} \mathtt{cs}$ *of* $\mathtt{st}$ *to events representing communications over channels in* $\mathtt{cs}$ *as follows.*

$$
\begin{array}{lcl}
\langle\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs} & = & \langle\rangle \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{E}} \mathtt{cs} & = & \langle \mathtt{d}.\alpha \rangle \frown (\mathtt{st} \upharpoonright_{\mathsf{E}} \mathtt{cs}),\ \textit{if}\ \mathtt{d} \in \mathtt{cs} \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{E}} \mathtt{cs} & = & \mathtt{st} \upharpoonright_{\mathsf{E}} \mathtt{cs},\ \textit{if}\ \mathtt{d} \notin \mathtt{cs}
\end{array}
$$

Definition 24. *For sets of constrained symbolic traces* $\mathtt{csts}_1$ *and* $\mathtt{csts}_2$ *with disjoint alphabets, and channel set* $\mathtt{cs}$*, we define the set of synchronising pairs* $\mathit{ScstraceP}_{\mathtt{cs}}(\mathtt{csts}_1, \mathtt{csts}_2)$ *as follows.*

$$
\begin{array}{l}
\mathit{ScstraceP}_{\mathtt{cs}}(\mathtt{csts}_1, \mathtt{csts}_2) = \\
\quad \{\, \mathtt{cst}_1 : \mathtt{csts}_1;\ \mathtt{cst}_2 : \mathtt{csts}_2 \mid (\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2) \,\}
\end{array}
$$

Definition 25. *For sets of constrained symbolic traces* $\mathtt{csts}_1$ *and* $\mathtt{csts}_2$*, channel set* $\mathtt{cs}$*, and alphabet* $\mathtt{a}$ *we define the set* $\mathit{Scstraces}^{\mathtt{a}}_{\mathtt{cs}}(\mathtt{csts}_1, \mathtt{csts}_2)$ *of synchronising constrained symbolic traces over* $\mathtt{a}$ *as follows.*

$$
\mathit{Scstraces}^{\mathtt{a}}_{\mathtt{cs}}(\mathtt{csts}_1, \mathtt{csts}_2) = \bigcup \{\, \mathtt{p} : \mathit{ScstraceP}_{\mathtt{cs}}(\mathtt{csts}_1, \mathtt{csts}_2) \bullet \mathcal{N}^{\mathtt{a}} \,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\, \}
$$

Definition 26.

$$
\mathtt{csts}_1 \ \mathrm{ccat}\ \mathtt{csts}_2 = \{\, \mathtt{st}_1, \mathtt{st}_2, \mathtt{c}_1, \mathtt{c}_2 \mid (\mathtt{st}_1, \mathtt{c}_1) \in \mathtt{csts}_1 \wedge (\mathtt{st}_2, \mathtt{c}_2) \in \mathtt{csts}_2 \bullet (\mathtt{st}_1 \frown \mathtt{st}_2, \mathtt{c}_1 \wedge \mathtt{c}_2) \,\}
$$

Definition 27. *For every constrained symbolic trace* $(\mathtt{st}, \mathtt{c})$ *and synchronisation channel set* $\mathtt{cs}$*, we define the projection* $(\mathtt{st}, \mathtt{c}) \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs}$ *of the trace* $(\mathtt{st}, \mathtt{c})$ *to channels not in* $\mathtt{cs}$ *as* $(\mathtt{st}, \mathtt{c}) \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs} = \mathcal{R}(\mathtt{st} \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs}, \mathtt{c})$*, where the projection operator for symbolic traces is defined inductively as follows.*

$$
\begin{array}{lcl}
\langle\rangle \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \langle\rangle \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \langle \mathtt{d}.\alpha \rangle \frown (\mathtt{st} \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs}),\ \textit{if}\ \mathtt{d} \notin \mathtt{cs} \\
(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}) \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs} & = & \mathtt{st} \upharpoonright\!\!\upharpoonright_{\mathsf{C}} \mathtt{cs},\ \textit{if}\ \mathtt{d} \in \mathtt{cs}
\end{array}
$$

Appendix B. Properties of the *Circus* operational semantics

The following propositions establish basic properties of the *Circus* operational semantics that we need. A simple property is captured by the proposition below. It states that, if the action does not involve a variable x, or more generally, any of the variables in a state $\mathtt{s}_3$, then its value in the state is not changed.

Proposition 1.

$$
(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xRightarrow{\mathtt{st}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)
$$

if, and only if,

$$
(\mathtt{c}_1 \mid \mathtt{s}_1;\ \mathtt{s}_3 \models \mathtt{A}_1) \xRightarrow{\mathtt{st}} (\mathtt{c}_2 \mid \mathtt{s}_2;\ \mathtt{s}_3 \models \mathtt{A}_2)
$$

provided $FV(\mathtt{s}_3) \cap FV(\mathtt{A}_1) = \varnothing$.

$FV(A)$ is the set of free variables of the action $\mathtt{A}$ as defined in [42]. In addition, if a constraint $\mathtt{c}_3$ is over

symbolic variables that are not used in the state nor in the symbolic trace, then it can be ignored.

Proposition 2.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$$

if, and only if,

$$(\mathtt{c}_1 \wedge \mathtt{c}_3 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \wedge \mathtt{c}_3 \mid \mathtt{s}_2 \models \mathtt{A}_2)$$

provided $FV(\mathtt{c}_3) \cap SV(\mathtt{s}_1, \mathtt{s}_2) = \varnothing$ *and* $FV(\mathtt{c}_3) \cap \alpha\mathtt{st} = \varnothing$.

We use $SV(s)$ to denote the set of symbolic variables used in the definition of $\mathtt{s}$ (or, more generally, in a list of states $\mathtt{s}_1, \mathtt{s}_2$ as used in the proposition above). By construction, constraints of a configuration are only ever strengthened.

Proposition 3.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$$

if, and only if, for some $\mathtt{c}_3$

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_1 \wedge \mathtt{c}_3 \mid \mathtt{s}_2 \models \mathtt{A}_2)$$

So, the constraint $\mathtt{c}_1$ used in an initial configuration is preserved in any configuration reached from it.

The operational semantics is consistent with the denotational semantics. So, for example, we have the property below for traces of an external choice.

Proposition 4.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \,\Box\, \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

for some $\mathtt{s}_2$ *and* $\mathtt{A}_3$ *if, and only if,*

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3) \quad \textit{or} \quad (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

This is consistent with the denotational semantics of $A_1 \,\Box\, A_2$, which defines that its set of traces is the union of those of A_1 and A_2. For sequential composition, we have the following.

Proposition 5.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \,;\, \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

for some $\mathtt{s}_2$ *and* $\mathtt{A}_3$ *if, and only if,*

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

or, there are $\mathtt{st}_1$, $\mathtt{c}'_1$, $\mathtt{s}'_1$, *and* $\mathtt{st}_2$ *such that* $\mathtt{st} = \mathtt{st}_1 \frown \mathtt{st}_2$,

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}'_1 \mid \mathtt{s}'_1 \models \mathtt{Skip}) \quad \textit{and} \quad (\mathtt{c}'_1 \mid \mathtt{s}'_1 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

The next property is concerned with the semantics of parallel actions. It relates the traces of a parallelism $\mathtt{A}_1 \,[\![\, \mathtt{x}_1 \mid \mathtt{cs} \mid \mathtt{x}_2 \,]\!]\, \mathtt{A}_2$ of actions, as calculated by the operational semantics, with the result of applying the synchronisation operator to the traces $\mathtt{st}_1$ and $\mathtt{st}_2$ of the parallel actions $\mathtt{A}_1$ and $\mathtt{A}_2$. This is needed in this work, where we consider parallel processes, because parallelism of processes is defined in terms of the parallelism of their main actions, with name sets defined by the states of the processes.

Proposition 6.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \llbracket \mathtt{x}_1 \mid \mathtt{cs} \mid \mathtt{x}_2 \rrbracket \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$$

for some $\mathtt{s}_2$ *and* $\mathtt{A}_3$ *if, and only if, there are* $\mathtt{st}_1$, $\mathtt{c}_1'$, $\mathtt{s}_1'$, $\mathtt{A}_1'$, $\mathtt{st}_2$, $\mathtt{c}_2'$, $\mathtt{s}_2'$, $\mathtt{A}_2'$, *and* $\mathtt{a}$, *such that* $\alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing$, $\alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1'$, $\alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2'$, $\alpha\mathtt{st} \leq \mathtt{a}$,

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_1{}' \mid \mathtt{s}_1' \models \mathtt{A}_1'), \quad (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2{}' \mid \mathtt{s}_2' \models \mathtt{A}_2')$$

and

$$\mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!| \mathcal{R} (\!| \mathtt{st}_1 \llbracket \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \rrbracket \mathtt{st}_2 |\!) |\!)$$

We need to apply $\mathcal{R}$ to all constrained symbolic traces: that obtained from the operational semantics of $\mathtt{A}_1 \llbracket \mathtt{x}_1 \mid \mathtt{cs} \mid \mathtt{x}_2 \rrbracket \mathtt{A}_2$ and those obtained by the synchronisation of $\mathtt{st}_1$ and $\mathtt{st}_2$. The renaming carried out by $\mathcal{N}^{\mathtt{a}}$ ensures that all symbolic traces are over the same alphabet $\mathtt{a}$, and so, can be compared.

Finally, we have a property of the operational semantics of hiding.

Proposition 7.

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A} \setminus \mathtt{cs}) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$$

for some $\mathtt{s}_2$ *and* $\mathtt{A}_3$ *if, and only if, there are* $\mathtt{st}_2$, $\mathtt{c}_3$, $\mathtt{s}_3$, $\mathtt{A}_3$, *and* $\mathtt{a}$, *such that* $\alpha\mathtt{st}_1 \leq \mathtt{a}$,

$$(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \textit{and} \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_2) = \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}_2, \mathtt{c}_3) \restriction_{\mathsf{C}} \mathtt{cs})$$

We consider traces $\mathtt{st}_2$ defined by the evaluation of $\mathtt{A}$, and their projections to remove the channels in $\mathtt{cs}$.

Appendix C. Some lemmas and their proofs

Lemma 1.

$$\mathit{Scstraces}_{\mathtt{cs}}^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1, \mathtt{P}_2) = \mathit{Scstraces}_{\mathtt{cs}}^{\mathtt{a},(\mathtt{a}_3,\mathtt{a}_4)}(\mathtt{P}_1, \mathtt{P}_2)$$

Proof. There is a bijection between $\mathit{cstraces}^{\mathtt{a}_1}(\mathtt{P}_1)$ and $\mathit{cstraces}^{\mathtt{a}_3}(\mathtt{P}_1)$ reflecting the fact that their symbolic traces characterise the same set of traces, namely those of $\mathtt{P}_1$, and the only difference between the symbolic traces in these sets is in their use of names of symbolic variables. The same comment applies to $\mathit{cstraces}^{\mathtt{a}_2}(\mathtt{P}_2)$ and $\mathit{cstraces}^{\mathtt{a}_4}(\mathtt{P}_2)$. We, therefore, can conclude that there is such a bijection between $\mathit{ScstraceP}_{\mathtt{cs}}^{(\mathtt{a}_1,\mathtt{a}_2)}(\mathtt{P}_1, \mathtt{P}_2)$ and $\mathit{Scstraces}_{\mathtt{cs}}^{(\mathtt{a}_3,\mathtt{a}_4)}(\mathtt{P}_1, \mathtt{P}_2)$. So, this lemma is a direct consequence of the definition of $\mathcal{N}^{\mathtt{a}}$. □

Appendix C.1. Properties of synchronisation

The constraints of all traces are the same.

Lemma 2.

$$\forall\, \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet \mathtt{cst}.2 = (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$$

Proof. By induction.

Case 1.

$$\begin{aligned}
&(\mathtt{st}, \mathtt{c}_1) \in \langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle\rangle \\
&\Rightarrow \mathtt{c}_1 = \mathtt{c}_2 && \text{[definition of } (\langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle\rangle)\text{]} \\
&\Leftrightarrow \mathtt{c}_1 = ((\langle\rangle \restriction_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle\rangle \restriction_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}_2 && \text{[definition of } \restriction_{\mathsf{E}} \text{ and } =_{\mathsf{S}}\text{]}
\end{aligned}$$

Case 2.

$$
\begin{array}{ll}
& (\mathtt{st}, \mathtt{c}_1) \in \langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \langle \mathtt{d}.\beta \rangle \\
\Rightarrow & \mathtt{c}_1 = \mathtt{c}_2 \qquad [\text{definition of } \langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \langle \mathtt{d}.\beta \rangle] \\
\Leftrightarrow & \mathtt{c}_1 = ((\langle\,\rangle \restriction_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \restriction_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}_2 \qquad [(\langle \mathtt{d}.\beta \rangle \restriction_{\mathsf{E}} \mathtt{cs}) = \langle\,\rangle]
\end{array}
$$

Case 3:. $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$$
\begin{array}{ll}
& (\mathtt{st}, \mathtt{c}_1) \in (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \\
\Rightarrow & \exists\, \mathtt{st}_3, \mathtt{c}_3 \bullet \\
& (\mathtt{st}_3, \mathtt{c}_3) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \mathtt{st}_2) \wedge \mathtt{st}_1 = \langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_3 \wedge \mathtt{c}_1 = \mathtt{c}_3 \\
& \qquad [\text{definition of } (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)] \\
\Leftrightarrow & \exists\, \mathtt{st}_3 \bullet (\mathtt{st}_3, \mathtt{c}_1) \in (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \mathtt{st}_2 \wedge \mathtt{st}_1 = \langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_3 \qquad [\text{predicate calculus}] \\
\Rightarrow & \mathtt{c}_1 = ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_2 \qquad [\text{induction hypothesis}] \\
\Leftrightarrow & \mathtt{c}_1 = ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_2 \qquad [(\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs} = \mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs}]
\end{array}
$$

Case 4:. $\mathtt{d} \in \mathtt{cs}$

$$
\begin{array}{ll}
& (\mathtt{st}, \mathtt{c}_1) \in (\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \\
\Leftrightarrow & \exists\, \mathtt{st}_3 \bullet (\mathtt{st}_3, \mathtt{c}_1) \in \mathtt{st}_1 \,[\![\, \mathtt{cs}, (\mathtt{c}_2 \wedge \alpha = \beta) \,]\!]\, \mathtt{st}_2 \wedge \mathtt{st}_1 = \langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_3 \\
& \qquad [\text{definition of } (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)] \\
\Rightarrow & \mathtt{c}_1 = (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs}) \wedge (\mathtt{c}_2 \wedge \alpha = \beta) \qquad [\text{induction hypothesis}] \\
\Leftrightarrow & \mathtt{c}_1 = (\langle \mathtt{d}.\alpha \rangle \frown (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} \langle \mathtt{d}.\beta \rangle \frown (\mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}_2 \qquad [\text{definition of } =_{\mathsf{S}}] \\
\Leftrightarrow & \mathtt{c}_1 = ((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_2 \\
& \qquad [(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} = \langle \mathtt{d}.\alpha \rangle \frown (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs}) \text{ and } (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs} = \langle \mathtt{d}.\beta \rangle \frown (\mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs})]
\end{array}
$$

Case 5:. $\mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$$
\begin{array}{ll}
& (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \\
\Rightarrow & \exists\, \mathtt{st}_3 \bullet \; (\mathtt{st}_3, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \mathtt{st}_2) \vee \\
& \qquad\quad (\mathtt{st}_3, \mathtt{c}_1) \in (\mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \\
& \qquad [\text{definition of } (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)] \\
\Rightarrow & \mathtt{c}_1 = ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_2 \vee \qquad [\text{induction hypothesis}] \\
& \mathtt{c}_1 = (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}_2 \\
\Rightarrow & \mathtt{c}_1 = ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}_2 \\
& \qquad [(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \restriction_{\mathsf{E}} \mathtt{cs} = (\mathtt{st}_1 \restriction_{\mathsf{E}} \mathtt{cs}) \text{ and } (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \restriction_{\mathsf{E}} \mathtt{cs} = (\mathtt{st}_2 \restriction_{\mathsf{E}} \mathtt{cs})]
\end{array}
$$

□

The projection of the synchronisation traces in $\mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2$ to their symbolic traces does not change if we quantify one of the variables (potentially) free in $\mathtt{c}$.

Lemma 3.

$$(\!|\, \mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2 \,|\!)\,.1 = (\!|\, \mathtt{st}_1 \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2 \,|\!)\,.1$$

Proof. By induction.

Case 1(a). : provided c

$$
\begin{aligned}
&(\!|\langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \langle\,\rangle |\!)\,.1 \\
&= (\!|\{\, (\langle\,\rangle, \mathtt{c}\,) \,\}|\!)\,.1 && \text{[synchronisation operator]} \\
&= \{\, \langle\,\rangle \,\} && \text{[relational image]} \\
&= (\!|\{\, (\langle\,\rangle, (\exists\, \mathtt{x} \bullet \mathtt{c})) \,\}|\!)\,.1 && \text{[relational image]} \\
&= (\!|\langle\,\rangle \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \langle\,\rangle |\!)\,.1 && \text{[synchronisation operator]}
\end{aligned}
$$

Case 1(b). : otherwise

$$
\begin{aligned}
&(\!|\langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \langle\,\rangle |\!)\,.1 \\
&= \varnothing && \text{[synchronisation operator and relational image]} \\
&= (\!|\langle\,\rangle \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \langle\,\rangle |\!)\,.1 && \text{[}(\exists\, \mathtt{x} \bullet \mathtt{c}) \text{ does not hold]}
\end{aligned}
$$

Case 2(a). : $(\langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, \langle \mathtt{d}.\beta \rangle)$, provided c

$$
\begin{aligned}
&(\!|\langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \langle \mathtt{d}.\beta \rangle |\!)\,.1 \\
&= (\!|\{\, (\langle \mathtt{d}.\beta \rangle, \mathtt{c}\,) \,\}|\!)\,.1 && \text{[synchronisation operator]} \\
&= \{\, \langle \mathtt{d}.\beta \rangle \,\} && \text{[relational image]} \\
&= (\!|\{\, (\langle \mathtt{d}.\beta \rangle, (\exists\, \mathtt{x} \bullet \mathtt{c})) \,\}|\!)\,.1 && \text{[relational image]} \\
&= (\!|\langle\,\rangle \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \langle \mathtt{d}.\beta \rangle |\!)\,.1 && \text{[synchronisation operator]}
\end{aligned}
$$

Case 2(a). : $(\langle\,\rangle \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \langle \mathtt{d}.\beta \rangle)$, provided c does not hold
Similar to Case 1(b).

Case 3:. $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$$
\begin{aligned}
&(\!|(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) |\!)\,.1 \\
&= (\!|\{\, \mathtt{cst} : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}|\!)\,.1 && \text{[synchronisation operator]} \\
&= (\!|\{\, \mathtt{cst}_1 : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2;\ \mathtt{cst}_2 : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2 \bullet \\
&\qquad (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}_1.1, \mathtt{cst}_2.2) \,\}|\!)\,.1 \\
&\qquad && \text{[induction hypothesis]} \\
&= \{\, \mathtt{cst} : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2 \bullet \langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}.1 \,\} && \text{[relational image]} \\
&= (\!|\{\, \mathtt{cst} : (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}|\!)\,.1 && \text{[relational image]} \\
&= (\!|(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) |\!)\,.1 && \text{[synchronisation operator]}
\end{aligned}
$$

Case 4:. $\mathtt{d} \in \mathtt{cs}$

$$
\begin{aligned}
&(\!|(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) |\!)\,.1 \\
&= (\!|\{\, \mathtt{cst} : \mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2 \bullet (\langle \mathtt{d}.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}|\!)\,.1 && \text{[synchronisation operator]} \\
&= (\!|\{\, \mathtt{cst}_1 : \mathtt{st}_1 \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2;\ \mathtt{cst}_2 : \mathtt{st}_1 \,[\![\, \mathtt{cs}, \mathtt{c} \,]\!]\, \mathtt{st}_2 \bullet && \text{[induction hypothesis]} \\
&\qquad (\langle \mathtt{d}.\beta \rangle \frown \mathtt{cst}_1.1, \mathtt{cst}_2.2) \,\}|\!)\,.1 \\
&= \{\, \mathtt{cst} : \mathtt{st}_1 \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2 \bullet \langle \mathtt{d}.\beta \rangle \frown \mathtt{cst}.1 \,\} && \text{[relational image]} \\
&= (\!|\{\, \mathtt{cst} : \mathtt{st}_1 \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, \mathtt{st}_2 \bullet (\langle \mathtt{d}.\beta \rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \,\}|\!)\,.1 && \text{[relational image]} \\
&= (\!|(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \,]\!]\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) |\!)\,.1 && \text{[synchronisation operator]}
\end{aligned}
$$

Case 5:. $(\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2)$, with $\mathtt{d}_1 \neq \mathtt{d}_2, \mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \in \mathtt{cs}$
Similar to Case 1(b).

Case 6:. $\mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$$
\begin{aligned}
&(\!|(\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_1.\beta\rangle \frown \mathtt{st}_2)|\!).1 \\
&= (\!| \{ \mathtt{cst} : (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \} \cup && \text{[synchronisation operator]} \\
&\quad \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2) \bullet (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \} |\!).1 \\
&= (\!|\{ \mathtt{cst} : (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \}|\!).1 \cup && \text{[property of relational image]} \\
&\quad (\!|\{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2) \bullet (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \}|\!).1 \\
&= (\!|\{ \mathtt{cst} : (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \}|\!).1 \cup && \text{[induction hypothesis]} \\
&\quad (\!|\{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2) \bullet (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \}|\!).1 \\
&= (\!| \{ \mathtt{cst} : (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \rrbracket \mathtt{st}_2 \bullet (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \} \cup \\
&\quad \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2) \bullet (\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{cst}.1, \mathtt{cst}.2) \} |\!).1 \\
&\qquad\qquad \text{[property of relational image]} \\
&= (\!|(\langle \mathtt{d}_1.\alpha\rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}) \rrbracket (\langle \mathtt{d}_2.\beta\rangle \frown \mathtt{st}_2)|\!).1 && \text{[synchronisation operator]}
\end{aligned}
$$

□

In the lemma below, we have a simple property of $\mathcal{R}$ when used in conjunction with the synchronisation operator. Basically, we can, in advance, consider a weaker constraint that already quantifies away variables that represent internal values: those that are not used in either of the traces $\mathtt{st}_1$ and $\mathtt{st}_2$ synchronised.

Lemma 4.

$$\mathcal{R}\,(\!|\, \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 |\!|) = \mathcal{R}\,(\!|\, \mathtt{st}_1 \llbracket \mathtt{cs}, (\exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \rrbracket \mathtt{st}_2 |\!|)$$

Proof.

$$
\begin{aligned}
&\mathcal{R}\,(\!|\, \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 |\!|) \\
&= \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet \mathcal{R}(\mathtt{cst}.1, \mathtt{cst}.2) \} && \text{[relational image]} \\
&= \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet \mathcal{R}(\mathtt{cst}.1, (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}) \} && \text{[Lemma 2]} \\
&= \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet \mathcal{R}(\mathtt{cst}.1, \exists\, \mathtt{x}_1 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}) \} \\
&\qquad\qquad \text{[definition of } \mathcal{R}\text{, with } \mathtt{x}_1 = \alpha(\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \mathtt{c}) \setminus \alpha\mathtt{cst}.1] \\
&= \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\mathtt{cst}.1, \exists\, \mathtt{x}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}) \} \\
&\qquad \text{[property of sets, with } \mathtt{x}_2 = (\alpha(\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}), \alpha\mathtt{c} \cap \alpha(\mathtt{st}_1, \mathtt{st}_2), \alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \setminus \alpha\mathtt{cst}.1] \\
&= \{ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\mathtt{cst}.1, \exists\, \mathtt{x}_3 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}) \} \\
&\qquad [\alpha\mathtt{cst}.1 \subseteq \alpha(\mathtt{st}_1, \mathtt{st}_2), \text{ with } \mathtt{x}_3 = (\alpha(\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}), \alpha\mathtt{c} \cap \alpha(\mathtt{st}_1, \mathtt{st}_2)) \setminus \alpha\mathtt{cst}.1, \alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)] \\
&= \{ \ \mathtt{cst} : \mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2 \bullet (\mathtt{cst}.1, \exists\, \mathtt{x}_4 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \} \\
&\qquad\qquad \text{[predicate calculus, with } \mathtt{x}_4 = (\alpha(\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}, \alpha\mathtt{c} \cap \alpha(\mathtt{st}_1, \mathtt{st}_2)) \setminus \alpha\mathtt{cst}.1]
\end{aligned}
$$

$= \{\ \mathtt{cst} : \mathtt{st}_1 \llbracket\, \mathtt{cs}, (\exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \,\rrbracket\, \mathtt{st}_2 \bullet$ [Lemma 3]
$\quad (\mathtt{cst}.1, \exists\, \mathtt{x}_4 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c})\,\}$

$= \{\ \mathtt{cst} : \mathtt{st}_1 \llbracket\, \mathtt{cs}, (\exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \,\rrbracket\, \mathtt{st}_2 \bullet$ [definition of $\mathcal{R}$]
$\quad \mathcal{R}(\mathtt{cst}.1, (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs} =_{\mathsf{S}} \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c})\,\}$

$= \{\, \mathtt{cst} : \mathtt{st}_1 \llbracket\, \mathtt{cs}, (\exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \,\rrbracket\, \mathtt{st}_2 \bullet \mathcal{R}(\mathtt{cst})\,\}$ [Lemma 2]

$= \mathcal{R}\,(\!|\, \mathtt{st}_1 \llbracket\, \mathtt{cs}, (\exists(\alpha\mathtt{c} \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)) \bullet \mathtt{c}) \,\rrbracket\, \mathtt{st}_2 \,|\!)$ [relational image]

□

The following lemma establishes that the set of traces $\mathtt{st}_1 \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \mathtt{st}_2$ is not empty exactly when $\mathtt{c}$ is satisfiable, even in the presence of the synchronisation requirements imposed by $\mathtt{st}_1$ and $\mathtt{st}_2$.

Lemma 5.

$$(\mathtt{st}_1 \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \mathtt{st}_2) \neq \varnothing \;\Leftrightarrow\; \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$$

where $\alpha\mathtt{st}_1 \leq \mathtt{a}_1$ *and* $\alpha\mathtt{st}_2 \leq \mathtt{a}_2$.

Proof. By induction.

Case 1.

$(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \langle\,\rangle) \neq \varnothing$
$\Leftrightarrow c$ [definition of $(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \langle\,\rangle)$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\langle\,\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle\,\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [$(\langle\,\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) = \langle\,\rangle$ and $\langle\,\rangle =_{\mathsf{S}} \langle\,\rangle = \mathtt{true}$]

Case 2(a). $: \mathtt{d} \notin \mathtt{cs}$

$(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \langle \mathtt{d}.\beta \rangle) \neq \varnothing$
$\Leftrightarrow c$ [definition of $(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c}) \,\rrbracket\, \langle \mathtt{d}.\beta \rangle)$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\langle\,\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [$\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{E}} \mathtt{cs} = \langle\,\rangle$]

Case 2(b). $: \mathtt{d} \in \mathtt{cs}$

$(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \langle \mathtt{d}.\beta \rangle) \neq \varnothing$
$\Leftrightarrow$ *false* [definition of $(\langle\,\rangle \llbracket\, \mathtt{cs}, \mathtt{c}) \,\rrbracket\, \langle \mathtt{d}.\beta \rangle)$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\langle\,\rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [$\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{E}} \mathtt{cs} = \langle \mathtt{d}.\beta \rangle$ and $\langle\,\rangle =_{\mathsf{S}} \langle \mathtt{d}.\beta \rangle = \mathtt{false}$]

Case 3:. $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \neq \varnothing$
$\Leftrightarrow ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, \mathtt{st}_2) \neq \varnothing$ [definition of $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [induction hypothesis]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [$((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) = (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs})$]

Case 4:. $\mathtt{d} \in \mathtt{cs}$

$((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)) \neq \varnothing$
$\Leftrightarrow (\mathtt{st}_1 \llbracket\, \mathtt{cs}, (\mathtt{c} \wedge \alpha = \beta) \,\rrbracket\, \mathtt{st}_2) \neq \varnothing$ [definition of $(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c} \,\rrbracket\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c} \wedge \alpha = \beta$ [induction hypothesis]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\langle \mathtt{d}.\alpha \rangle \frown (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs})) =_{\mathsf{S}} (\langle \mathtt{d}.\beta \rangle \frown (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs})) \wedge \mathtt{c}$ [definition of $=_{\mathsf{S}}$]
$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$ [definition of $\upharpoonright_{\mathsf{E}}$]

Case 5:. $\mathtt{d}_1 \neq \mathtt{d}_2, \mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \in \mathtt{cs}$

$((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \neq \varnothing$

$\Leftrightarrow$ *false* [definition of $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$

$[((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}_2.\alpha \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs})$ is $\mathtt{false}]$

Case 6:. $\mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \neq \varnothing$

$\Leftrightarrow ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket \mathtt{st}_2) \neq \varnothing \vee (\mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \neq \varnothing$

[definition of $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c} \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow$ $(\exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}) \vee$ [induction hypothesis]
$(\exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c})$

$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}$

$[(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) = (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs})$ and $((\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) = (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs})]$

□

With the next lemma, we establish that, if the projection of both traces $\mathtt{st}_1$ and $\mathtt{st}_2$ to synchronisation channels is nonempty, then the projection of all synchronisation traces to these channels is nonempty.

Lemma 6.

$$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in (\mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \mathtt{st}_2) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$$

provided $(\mathtt{st}_1 \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \mathtt{st}_2) \neq \varnothing$.

Proof. By induction.

Case 1.

$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in (\langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle\rangle) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$

$\Leftrightarrow$ *false* [definitions of $(\langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle\rangle)$ and $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow \langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \wedge \langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$ $[\langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} = \langle\rangle]$

Case 2. : $\mathtt{d} \notin \mathtt{cs}$

$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in (\langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle \mathtt{d}.\beta \rangle) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$

$\Leftrightarrow$ *false* [definition of $(\langle\rangle \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \langle \mathtt{d}.\beta \rangle)$ and $\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} = \langle\rangle$, if $\mathtt{d} \notin \mathtt{cs}$]

$\Leftrightarrow \langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \wedge \langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$ $[\langle\rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} = \langle\rangle$ (and $\langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathsf{C}} \mathtt{cs} = \langle\rangle)]$

Case 3:. $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \mathtt{st}_2) \Rightarrow (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$

[definition of $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket \mathtt{st}_2) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$ [induction hypothesis]

$\Leftrightarrow (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle \wedge (\langle \mathtt{d}_2.\alpha \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$]

Case 4:. $\mathtt{d} \in \mathtt{cs}$

$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in (\mathtt{st}_1 \llbracket\, \mathtt{cs}, (\mathtt{c}_2 \wedge \alpha = \beta) \,\rrbracket\, \mathtt{st}_2) \Rightarrow (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

[definition of $(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow$ *true* [$(\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ by definition of $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow (\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$]

Case 5:. $\mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow$ $(\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket\, \mathtt{st}_2) \Rightarrow (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle) \wedge$
$(\forall\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}_1) \in (\mathtt{st}_1 \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \Rightarrow (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$

[definition of $((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket\, \mathtt{cs}, \mathtt{c}_2 \,\rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2))$]

$\Leftrightarrow$ $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge$ [induction hypothesis]
$\mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$]

□

The following lemmas lift to the synchronisation operator for constrained symbolic traces results similar to some previously established for the operator for symbolic traces.

Lemma 7.

$$\forall\, \mathtt{cst} \bullet \mathtt{cst} \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{cst}_1 \llbracket\, \mathtt{cs} \,\rrbracket\, \mathtt{cst}_2 \,|\!) \Rightarrow \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \Leftrightarrow \mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$$

where $\alpha\mathtt{st}_1 \leq \mathtt{a}_1$ *and* $\alpha\mathtt{st}_2 \leq \mathtt{a}_2$*, and provided* $\mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{cst}_1 \llbracket\, \mathtt{cs} \,\rrbracket\, \mathtt{cst}_2 \,|\!) \neq \varnothing$*.*

Proof.

$\forall\, \mathtt{cst} \bullet \mathtt{cst} \in \mathcal{N}^{\mathtt{a}} (\!|\, (\mathtt{st}_1, \mathtt{c}) \llbracket\, \mathtt{cs} \,\rrbracket\, (\mathtt{st}_2, \mathtt{c}) \,|\!) \Rightarrow \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\mathtt{st}_1, \mathtt{c}_1) \llbracket\, \mathtt{cs} \,\rrbracket\, (\mathtt{st}_2, \mathtt{c}_2)) \Rightarrow (\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\mathcal{N}$]

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \llbracket\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,\rrbracket\, \mathtt{st}_2) \Rightarrow (\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

[definition of synchronising operator]

$\Leftrightarrow \forall\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \llbracket\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,\rrbracket\, \mathtt{st}_2) \Rightarrow \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

[definition of $\upharpoonright_{\mathsf{C}}$ for constrained symbolic traces]

$\Leftrightarrow \mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [Lemma 6 and proviso]

$\Leftrightarrow (\mathtt{st}_1, \mathtt{c}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\mathtt{st}_2, \mathtt{c}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$ for constrained symbolic traces]

□

Lemma 8.

$\exists\, \mathtt{cst} \bullet \mathtt{cst} \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{cst}_1 \llbracket\, \mathtt{cs} \,\rrbracket\, \mathtt{cst}_2 \,|\!) \wedge \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$
$\Rightarrow$
$\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2)$

Proof.

$\exists\, \mathtt{cst} \bullet \mathtt{cst} \in \mathcal{N}^{\mathtt{a}} (\!|\, (\mathtt{st}_1, \mathtt{c}_1) [\![\, \mathtt{cs} \,]\!] (\mathtt{st}_2, \mathtt{c}_2) \,|\!) \wedge \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow \exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\mathtt{st}_1, \mathtt{c}_1) [\![\, \mathtt{cs} \,]\!] (\mathtt{st}_2, \mathtt{c}_2)) \wedge (\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\mathcal{N}$]

$\Leftrightarrow \exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!] \mathtt{st}_2) \wedge (\mathtt{st}, \mathtt{c}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of synchronisation operator]

$\Leftrightarrow \exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!] \mathtt{st}_2) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$
[definition of $\upharpoonright_{\mathsf{C}}$ for constrained symbolic traces]

$\Rightarrow \mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \wedge \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$ [Lemma 11]

$\Leftrightarrow (\mathtt{st}_1, \mathtt{c}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\mathtt{st}_2, \mathtt{c}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \wedge \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$
[definition of $\upharpoonright_{\mathsf{C}}$ for constrained symbolic traces]

□

The lemma below establishes that the traces built from the (local) traces of $\mathtt{P}_1$ and $\mathtt{P}_2$, are all those of the parallelism between $\mathtt{P}_1$ and $\mathtt{P}_2$.

Lemma 9. *For every pair of disjoint alphabets* $\mathtt{a}_1$ *and* $\mathtt{a}_2$

$$\mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1 [\![\, \mathtt{cs} \,]\!] \mathtt{P}_2) = \bigcup \{\, \mathtt{cst}_1 : \mathit{cstraces}^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{a}_2}(\mathtt{P}_2) \bullet \mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{cst}_1 [\![\, \mathtt{cs} \,]\!] \mathtt{cst}_2 \,|\!) \,\}$$

Proof. In giving the semantics of $\mathtt{P}_1 [\![\, \mathtt{cs} \,]\!] \mathtt{P}_2$, we observe that we can assume, without loss of generality, that

$$\mathtt{P}_1 = \mathtt{begin\ state}\ [\mathtt{x}_1 : \mathtt{T}_1] \bullet \mathtt{A}_1\ \mathtt{end} \text{ and } \mathtt{P}_2 = \mathtt{begin\ state}\ [\mathtt{x}_2 : \mathtt{T}_2] \bullet \mathtt{A}_2\ \mathtt{end}$$

with $\mathtt{x}_1 \cap \mathtt{x}_2 = \varnothing$, where $\mathtt{x}_1$ and $\mathtt{x}_2$ are the lists of state components of $\mathtt{P}_1$ and $\mathtt{P}_2$, and for simplicity, we sometimes use $\mathtt{x}_1$ and $\mathtt{x}_2$ as sets of variable names.

$\mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1 [\![\, \mathtt{cs} \,]\!] \mathtt{P}_2)$

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{begin\ state}\ [\mathtt{x}_1 : \mathtt{T}_1;\ \mathtt{x}_2 : \mathtt{T}_2] \bullet \mathtt{A}_1 [\![\, \mathtt{x}_1 \mid \mathtt{cs} \mid \mathtt{x}_2 \,]\!] \mathtt{A}_2 \mathtt{end})$ [semantics of process parallelism]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2, \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2, \mathtt{A}_1 \parallel \mathtt{A}_2)$
[definition of *cstraces* for processes and abbreviation $\parallel$ for $[\![x_1 \mid cs \mid x_2]\!]$]

$= \{\, \mathtt{st}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1 \parallel \mathtt{A}_2) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_3)$ [definition of *cstraces* for actions]
$\bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \,\}$

$= \{\, \mathtt{st}, \mathtt{c}_2 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge$ [Proposition 6]
$\exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{s}_1', \mathtt{A}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{s}_2', \mathtt{A}_2', \mathtt{b} \bullet$
$(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \overset{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_1{}' \mid \mathtt{s}_1' \models \mathtt{A}_1') \wedge$
$(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_2) \overset{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2{}' \mid \mathtt{s}_2' \models \mathtt{A}_2') \wedge$
$\alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \alpha\mathtt{st} \leq \mathtt{b} \wedge$
$\mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{b}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!)$
$\bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \,\}$

$= \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{s}_1', \mathtt{A}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{s}_2', \mathtt{A}_2' \bullet$ [Lemma 13 and $\alpha\mathtt{st} \leq \mathtt{a}$ by definition of $\mathcal{N}^{\mathtt{a}}$ and $\mathcal{R}$]
$(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \overset{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_1{}' \mid \mathtt{s}_1' \models \mathtt{A}_1') \wedge$
$(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_2) \overset{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2{}' \mid \mathtt{s}_2' \models \mathtt{A}_2') \wedge$
$\alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2'$
$\mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!)$
$\bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \,\}$

$$
\begin{array}{ll}
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{s}_1', \mathtt{A}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{s}_2', \mathtt{A}_2' \bullet \qquad \text{[Proposition 1]} \\
& \quad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1 := \mathtt{w}_1 \models \mathtt{A}_1) \overset{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_1{}' \mid \mathtt{s}_1' \models \mathtt{A}_1') \wedge \\
& \quad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{A}_2) \overset{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2{}' \mid \mathtt{s}_2' \models \mathtt{A}_2') \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{s}_1', \mathtt{A}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{s}_2', \mathtt{A}_2' \bullet \qquad \text{[Proposition 2]} \\
& \quad (\mathtt{w}_1 \in \mathtt{T}_1 \mid \mathtt{x}_1 := \mathtt{w}_1 \models \mathtt{A}_1) \overset{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_1{}' \mid \mathtt{s}_1' \models \mathtt{A}_1') \wedge \\
& \quad (\mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{A}_2) \overset{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2{}' \mid \mathtt{s}_2' \models \mathtt{A}_2') \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[definition of } \mathit{cstraces}\text{]} \\
& \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') \in \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \wedge \\
& \quad \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \in \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[Lemma 4, with } \mathtt{x} = \alpha(\mathtt{c}_1', \mathtt{c}_2') \setminus \alpha(\mathtt{st}_1, \mathtt{st}_2)\text{]} \\
& \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') \in \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \wedge \\
& \quad \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \in \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, (\exists\, \mathtt{x} \bullet \mathtt{c}_1{}' \wedge \mathtt{c}_2{}') \,]\!] \mathtt{st}_2 \,|\!) \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[}\alpha\mathtt{c}_1' \cap \alpha\mathtt{c}_2' = \varnothing\text{, with } \mathtt{x}_1 = \alpha\mathtt{c}_1' \setminus \alpha\mathtt{st}_1 \text{ and } \mathtt{x}_2 = \alpha\mathtt{c}_2' \setminus \alpha\mathtt{st}_2\text{]} \\
& \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') \in \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \wedge \\
& \quad \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \in \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R} (\!|\, \mathtt{st}_1 [\![\, \mathtt{cs}, ((\exists\, \mathtt{x}_1 \bullet \mathtt{c}_1') \wedge (\exists\, \mathtt{x}_2 \bullet \mathtt{c}_2')) \,]\!] \mathtt{st}_2 \,|\!) \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[definition of synchronisation operator]} \\
& \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') \in \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \wedge \\
& \quad \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \in \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, (\mathtt{st}_1, \exists\, \mathtt{x}_1 \bullet \mathtt{c}_1') [\![\, \mathtt{cs} \,]\!] (\mathtt{st}_2, \exists\, \mathtt{x}_2 \bullet \mathtt{c}_2') \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{st}_1, \mathtt{c}_1', \mathtt{st}_2, \mathtt{c}_2', \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[definition of } \mathcal{R}\text{]} \\
& \quad \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') \in \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \wedge \\
& \quad \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \in \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \wedge \\
& \quad \alpha\mathtt{c}_1{}' \cap \alpha\mathtt{c}_2{}' = \varnothing \wedge \alpha\mathtt{st}_1 \subseteq \alpha\mathtt{c}_1' \wedge \alpha\mathtt{st}_2 \subseteq \alpha\mathtt{c}_2' \wedge \\
& \quad \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathcal{R}(\mathtt{st}_1, \mathtt{c}_1') [\![\, \mathtt{cs} \,]\!] \mathcal{R}(\mathtt{st}_2, \mathtt{c}_2') \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\} \\
= & \{\, \mathtt{st}, \mathtt{c}_2 \mid \exists\, \mathtt{b}_1, \mathtt{b}_2 \bullet \qquad \text{[predicate calculus and property of } \mathit{cstraces}\text{]} \\
& \quad \exists\ \mathtt{cst}_1 : \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1);\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \bullet \\
& \qquad \mathrm{disjoint}(\mathtt{b}_1, \mathtt{b}_2) \wedge \mathcal{R}(\mathtt{st}, \mathtt{c}_2) \in \mathcal{N}^{\mathtt{a}} (\!|\, \mathtt{cst}_1 [\![\, \mathtt{cs} \,]\!] \mathtt{cst}_2 \,|\!) \\
& \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\}
\end{array}
$$

$$= \bigcup\{\ \mathtt{b}_1, \mathtt{b}_2, \mathtt{cst}_1 : \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1);\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2) \mid$$
$$\text{disjoint}(\mathtt{b}_1, \mathtt{b}_2)$$
$$\bullet\ \mathcal{N}^{\mathtt{a}}\ (\!|\ \mathtt{cst}_1 \ [\![\ \mathtt{cs}\ ]\!]\ \mathtt{cst}_2\ |\!)\}$$

[property of sets and relational image]

$$= \bigcup\{\ \mathtt{cst}_1 : \mathit{cstraces}^{\mathtt{b}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1);\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{b}_2}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2)$$
$$\bullet\ \mathcal{N}^{\mathtt{a}}\ (\!|\ \mathtt{cst}_1 \ [\![\ \mathtt{cs}\ ]\!]\ \mathtt{cst}_2\ |\!)\}$$

[Lemma 12 and assumption: disjoint$(\mathtt{b}_1, \mathtt{b}_2)$]

□

Appendix C.2. Properties of satisfiability

Satiisfiability, as captured by $\mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$, does not ensure that there is an actual synchronisation arising from the traces $\mathtt{st}_1$ and ($\mathtt{st}_2$, since their projections ($\mathtt{st}_1 \upharpoonright_{\mathtt{E}} \mathtt{cs}$) and ($\mathtt{st}_2 \upharpoonright_{\mathtt{E}} \mathtt{cs}$) might be empty. It does ensure, however, that the set of traces defined by the synchronisation operator is not empty. This is established by the Lemma 10 proved below.

Lemma 10.

$$(\mathtt{st}_1 \ [\![\ \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2)\ ]\!]\ \mathtt{st}_2) \neq \varnothing \Leftrightarrow \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$$

Proof.

$$(\mathtt{st}_1 \ [\![\ \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2)\ ]\!]\ \mathtt{st}_2) \neq \varnothing$$
$$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathtt{E}} \mathtt{cs}) =_{\mathtt{S}} (\mathtt{st}_2 \upharpoonright_{\mathtt{E}} \mathtt{cs}) \wedge \mathtt{c}_1 \wedge \mathtt{c}_2 \qquad \text{[Lemma 5, with } \alpha\mathtt{st}_1 \leq \mathtt{a}_1 \text{ and } \alpha\mathtt{st}_2 \leq \mathtt{a}_2\text{]}$$
$$\Leftrightarrow \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2)) \qquad \text{[definition of } \mathit{satisfiable}\text{]}$$

□

This relationship between the notion of satisfiability and the existence of synchronisation traces reassures us of the adequacy of the definition of satisfiability.

The following lemma identifies a stronger condition that holds when the set of synchronisation traces not only is not empty, but also contains at least one trace with at least one synchronisation.

Lemma 11.

$$\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \ [\![\ \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2)\ ]\!]\ \mathtt{st}_2) \wedge \mathtt{st} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle$$
$$\Rightarrow$$
$$\mathtt{st}_1 \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$$

Proof. By induction.

Case 1.

$$\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\langle\,\rangle \ [\![\ \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2)\ ]\!]\ \langle\,\rangle) \wedge \mathtt{st} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle$$
$$\Leftrightarrow \mathit{false} \qquad \text{[definitions of } (\langle\,\rangle \ [\![\ \mathtt{cs}, \mathtt{c}_2\ ]\!]\ \langle\,\rangle) \text{ and } \upharpoonright_{\mathtt{C}}\text{]}$$
$$\Leftrightarrow \langle\,\rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \langle\,\rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle\,\rangle, \mathtt{c}_1), (\langle\,\rangle, \mathtt{c}_2)) \qquad [\langle\,\rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} = \langle\,\rangle]$$

Case 2. : $\mathtt{d} \notin \mathtt{cs}$

$$\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\langle\,\rangle \ [\![\ \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2)\ ]\!]\ \langle \mathtt{d}.\beta \rangle) \wedge \mathtt{st} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle$$
$$\Leftrightarrow \mathit{false} \qquad \text{[definition of } (\langle\,\rangle \ [\![\ \mathtt{cs}, \mathtt{c}_2\ ]\!]\ \langle \mathtt{d}.\beta \rangle) \text{ and } \langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} = \langle\,\rangle, \text{ if } \mathtt{d} \notin \mathtt{cs}\text{]}$$
$$\Leftrightarrow \langle\,\rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle\,\rangle, \mathtt{c}_1), (\langle \mathtt{d}.\beta \rangle, \mathtt{c}_2)) \quad [\langle\,\rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} = \langle\,\rangle \text{ (and } \langle \mathtt{d}.\beta \rangle \upharpoonright_{\mathtt{C}} \mathtt{cs} = \langle\,\rangle)]$$

Case 3:. $\mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$(\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$

$\Leftrightarrow (\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!]\, \mathtt{st}_2) \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$
[definition of $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow \exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!]\, \mathtt{st}_2) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ [definition of $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow\ (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$
[induction hypothesis]

$\Leftrightarrow\ (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\alpha \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))$
[definition of $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow\ (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\alpha \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge$ [definitions of *satisfiable* and $\upharpoonright_{\mathsf{E}}$]
$\mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}_2.\alpha \rangle \frown \mathtt{st}_2, \mathtt{c}_2))$

Case 4:. $\mathtt{d} \in \mathtt{cs}$

$\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!]\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$

$\Leftrightarrow (\exists\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2 \wedge \alpha = \beta) \,]\!]\, \mathtt{st}_2) \wedge (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$
[definition of $(\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2)$]

$\Leftrightarrow \exists\, \mathtt{st}, \mathtt{c}_1 \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2 \wedge \alpha = \beta) \,]\!]\, \mathtt{st}_2)$ [$(\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle$ by definition of $\upharpoonright_{\mathsf{C}}$]

$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet (\mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_1 \wedge \mathtt{c}_2 \wedge \alpha = \beta$ [Lemma 5, with $\alpha \mathtt{st}_1 \leq \mathtt{a}_1$ and $\alpha \mathtt{st}_2 \leq \mathtt{a}_2$]

$\Leftrightarrow \exists\, \mathtt{a}_1, \mathtt{a}_2 \bullet ((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} ((\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs}) \wedge \mathtt{c}_1 \wedge \mathtt{c}_2$ [definitions of $\upharpoonright_{\mathsf{E}}$ and $=_{\mathsf{S}}$]

$\Leftrightarrow \mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2, \mathtt{c}_2))$ [definition of *satisfiable*]

$\Leftrightarrow\ (\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}.\beta \rangle \frown \mathtt{st}_2, \mathtt{c}_2))$
[definition of $\upharpoonright_{\mathsf{C}}$]

Case 5:. $\mathtt{d}_1 \neq \mathtt{d}_2, \mathtt{d}_1 \in \mathtt{cs}, \mathtt{d}_2 \in \mathtt{cs}$

$(\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$

$\Leftrightarrow \mathit{false}$ [$(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \,[\![\, \mathtt{cs}, \mathtt{c}_2 \,]\!]\, (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) = \varnothing$]

$\Leftrightarrow\ (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge$
$\mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2, \mathtt{c}_2))$
[definition of *satisfiable* and $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}) =_{\mathsf{S}} (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs})$ is $\mathtt{false}$]

Case 6:. $\mathtt{d}_1 \notin \mathtt{cs}, \mathtt{d}_2 \notin \mathtt{cs}$

$(\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \wedge \mathtt{st} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$

$\Leftrightarrow$ $(\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}_1) \in ((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \rrbracket \mathtt{st}_2) \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$
$\vee$
$(\exists\, \mathtt{st}, \mathtt{c} \bullet (\mathtt{st}, \mathtt{c}) \in (\mathtt{st}_1 \llbracket \mathtt{cs}, (\mathtt{c}_1 \wedge \mathtt{c}_2) \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2)) \wedge (\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle)$

[definition of $((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \llbracket \mathtt{cs}, \mathtt{c}_2 \rrbracket (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2))$]

$\Leftrightarrow$ $((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\mathtt{st}_2, \mathtt{c}_2))) \vee$
$(\mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}((\mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2, \mathtt{c}_2)))$

[induction hypothesis]

$\Leftrightarrow$ $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \wedge$
$\mathit{satisfiable}_{\mathtt{cs}}((\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1, \mathtt{c}_1), (\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2, \mathtt{c}_2))$

$[(\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{C}} \mathtt{cs} = \mathtt{st}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}$ and $(\langle \mathtt{d}_2.\beta \rangle \frown \mathtt{st}_2) \upharpoonright_{\mathsf{E}} \mathtt{cs} = \mathtt{st}_2 \upharpoonright_{\mathsf{E}} \mathtt{cs}$, and]

$[(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{C}} \mathtt{cs} = \mathtt{st}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}$ and $(\langle \mathtt{d}_1.\alpha \rangle \frown \mathtt{st}_1) \upharpoonright_{\mathsf{E}} \mathtt{cs} = \mathtt{st}_1 \upharpoonright_{\mathsf{E}} \mathtt{cs}]$

□

The reverse implication is a direct consequence of Lemmas 10 and 6.

Appendix C.3. Properties of normalisation

When applied to equivalent constrained symbolic traces, $\mathcal{N}$ makes them syntactically equal.

Lemma 12.

$$\mathtt{cst}_1 \equiv \mathtt{cst}_2 \Rightarrow \mathcal{N}^{\mathtt{a}}(\mathtt{cst}_1) = \mathcal{N}^{\mathtt{a}}(\mathtt{cst}_2)$$

Proof. Direct from the definition of $\mathcal{N}$ and equivalence of constrained symbolic traces: $(\mathtt{st}_1, \mathtt{c}_1) \equiv (\mathtt{st}_2, \mathtt{c}_2)$ if and only if $\mathtt{st}_1 \upharpoonright_{\mathsf{C}} = \mathtt{st}_1 \upharpoonright_{\mathsf{C}}$ and $c_1[\alpha\mathtt{st}_2/\alpha\mathtt{st}_2] = c_2$. The projection operation $\mathtt{st} \upharpoonright_{\mathsf{C}}$ keeps just the channels of the trace $\mathtt{st}$. □

The following lemma gives a property of alphabets and $\mathcal{N}$.

Lemma 13. *For every* $\mathtt{a}_1$ *and* $\mathtt{a}_2$ *such that* $\alpha\mathtt{st} \leq \mathtt{a}_1$ *and* $\alpha\mathtt{st} \leq \mathtt{a}_2$,

$$(\mathtt{st}, \mathtt{c}) = \mathcal{N}^{\mathtt{a}_1}(\mathtt{cst}) \Leftrightarrow (\mathtt{st}, \mathtt{c}) = \mathcal{N}^{\mathtt{a}_2}(\mathtt{cst})$$

Proof.

$(\mathtt{st}, \mathtt{c}) = \mathcal{N}^{\mathtt{a}_1}(\mathtt{st}_1, \mathtt{c}_1)$

$\Leftrightarrow (\mathtt{st} = \mathtt{st}_1[\mathtt{a}_1/\alpha\mathtt{st}]) \wedge (\mathtt{c} = \mathtt{c}_1[\mathtt{a}_1/\alpha\mathtt{st}])$ [definition of $\mathcal{N}^{\mathtt{a}_1}$]

$\Leftrightarrow (\mathtt{st} = \mathtt{st}_1[\mathtt{a}_2/\alpha\mathtt{st}]) \wedge (\mathtt{c} = \mathtt{c}_1[\mathtt{a}_2/\alpha\mathtt{st}])$ [$\alpha\mathtt{st} \leq \mathtt{a}_1$ and $\alpha\mathtt{st} \leq \mathtt{a}_2$ imply $(\mathtt{a}_1 \restriction \#\,\mathtt{st}) \leq \mathtt{a}_2$]

$\Leftrightarrow (\mathtt{st}, \mathtt{c}) = \mathcal{N}^{\mathtt{a}_2}(\mathtt{st}_1, \mathtt{c}_1)$ [definition of $\mathcal{N}^{\mathtt{a}_2}$]

We use $(\mathtt{a} \restriction \#\, n)$ to refer to the first n elements of the alphabet $\mathtt{a}$, and $\#\,\mathtt{st}$ for the size of a trace $\mathtt{st}$. □

Appendix D. Proofs of theorems

i

Theorem 2.

$$Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2) = \{\, \mathtt{cst} : cstraces^{\mathtt{a}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \,\}$$

Proof.

$Scstraces^{\mathtt{a},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2)$

$= \bigcup\{\, \mathtt{p} : ScstraceP^{(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2) \bullet \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\}$ [definition of *Scstraces*]

$= \bigcup\{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2) \mid$ [definition of *ScstraceP*]
$(\mathtt{cst}_1 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge satisfiable_{\mathtt{cs}}(\mathtt{cst}_1,\mathtt{cst}_2)$
$\bullet\ \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!)\,\}$

$= \{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2);\ \mathtt{cst} : \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!) \mid$ [property of sets]
$(\mathtt{cst}_1 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge satisfiable_{\mathtt{cs}}(\mathtt{cst}_1,\mathtt{cst}_2)$
$\bullet\ \mathtt{cst}\,\}$

$= \{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2);\ \mathtt{cst} : \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!) \mid$ [Lemmas 10 and 7]
$(\mathtt{cst}_1 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge satisfiable_{\mathtt{cs}}(\mathtt{cst}_1,\mathtt{cst}_2) \wedge (\mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle$
$\bullet\ \mathtt{cst}\,\}$

$= \{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2);\ \mathtt{cst} : \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!) \mid (\mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle$ [Lemma 8]
$\bullet\ \mathtt{cst}\,\}$

$= \{\ \mathtt{cst} : \bigcup\{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2) \bullet \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!)\,\} \mid$ [property of sets]
$(\mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle\,\}$

$= \{\, \mathtt{cst} : cstraces^{\mathtt{a}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \,\}$ [Lemma 9]

□

Theorem 3.

$$Scstraces^{\mathtt{a}}_{\mathtt{cs}}(Scstraces^{\mathtt{a}_{12}}_{\mathtt{cs}}(cstraces^{\mathtt{a}_1}(\mathtt{P}_1), cstraces^{\mathtt{a}_2}(\mathtt{P}_2)), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$$
$$=$$
$$Scstraces^{\mathtt{a}}_{\mathtt{cs}}(cstraces^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$$

Proof.

$Scstraces^{\mathtt{a}}_{\mathtt{cs}}(Scstraces^{\mathtt{a}_{12}}_{\mathtt{cs}}(cstraces^{\mathtt{a}_1}(\mathtt{P}_1), cstraces^{\mathtt{a}_2}(\mathtt{P}_2)), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$

$= Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\bigcup\{\, \mathtt{p} : ScstraceP_{\mathtt{cs}}(cstraces^{\mathtt{a}_1}(\mathtt{P}_1), cstraces^{\mathtt{a}_2}(\mathtt{P}_2)) \bullet \mathcal{N}^{\mathtt{a}12}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\}, cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$
[Definition 25]

$= Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\ \bigcup\{\ \mathtt{cst}_1 : cstraces^{\mathtt{a}_1}(\mathtt{P}_1);\ \mathtt{cst}_2 : cstraces^{\mathtt{a}_2}(\mathtt{P}_2) \mid$ [Definition 24]
$(\mathtt{cst}_1 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathtt{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge satisfiable_{\mathtt{cs}}(\mathtt{cst}_1,\mathtt{cst}_2)$
$\bullet\ \mathcal{N}^{\mathtt{a}12}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!)\,\},$
$cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$

$= Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\bigcup\{\mathtt{p} : ScstraceP^{(\mathtt{a1},\mathtt{a2})}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2) \bullet \mathcal{N}^{\mathtt{a}12}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\}, cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$ [Definition 12]

$= Scstraces^{\mathtt{a}}_{\mathtt{cs}}(Scstraces^{\mathtt{a}_{12},(\mathtt{a}_1,\mathtt{a}_2)}_{\mathtt{cs}}(\mathtt{P}_1,\mathtt{P}_2), cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$ [Definition 15]

$= Scstraces^{\mathtt{a}}_{\mathtt{cs}}(\{\, \mathtt{cst} : cstraces^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \,\}, cstraces^{\mathtt{a}_3}(\mathtt{P}_3))$ [Theorem 2]

$= \bigcup\{\ \mathtt{p} : ScstraceP_{\mathtt{cs}}(\ \{\, \mathtt{cst} : cstraces^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathtt{C}} \mathtt{cs} \neq \langle\,\rangle \,\}, cstraces^{\mathtt{a}_3}(\mathtt{P}_3)) \bullet$
$\mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\}$

[definition of *Scstraces*]

$$= \bigcup\{\ \mathtt{p} : \{\quad \mathtt{cst}_1 : \{\, \mathtt{cst} : \mathit{cstraces}^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \,\};\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{a}_3}(\mathtt{P}_3)) \mid$$
$$(\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2)\,\} \bullet$$
$$\mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\}$$

[definition of *ScstraceP*]

$$= \bigcup\{\quad \mathtt{cst}_1 : \{\, \mathtt{cst} : \mathit{cstraces}^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2) \mid \mathtt{cst} \upharpoonright_{\mathsf{C}} \mathtt{cs} \neq \langle\,\rangle \,\};\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{a}_3}(\mathtt{P}_3)) \mid \quad \text{[property of sets]}$$
$$(\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2) \bullet$$
$$\mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!)\,\}$$

$$= \bigcup\{\quad \mathtt{cst}_1 : \mathit{cstraces}^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2);\ \mathtt{cst}_2 : \mathit{cstraces}^{\mathtt{a}_3}(\mathtt{P}_3)) \mid \quad \text{[property of sets]}$$
$$(\mathtt{cst}_1 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge (\mathtt{cst}_2 \upharpoonright_{\mathsf{C}} \mathtt{cs}) \neq \langle\,\rangle \wedge \mathit{satisfiable}_{\mathtt{cs}}(\mathtt{cst}_1, \mathtt{cst}_2) \bullet$$
$$\mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{cst}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{cst}_2 \,|\!)\,\}$$

$$= \bigcup\{\, \mathtt{p} : \mathit{ScstraceP}^{(\mathtt{a}_{12},\mathtt{a}_3)}_{\mathtt{cs}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2, \mathtt{P}_3) \mid \mathcal{N}^{\mathtt{a}}\,(\!|\, \mathtt{p}.1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{p}.2 \,|\!)\,\} \quad \text{[definition of } \mathit{ScstraceP}\text{]}$$

$$= \mathit{Scstraces}^{(\mathtt{a}_{12},\mathtt{a}_3)}_{\mathtt{cs}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2, \mathtt{P}_3) \quad \text{[definition of } \mathit{Scstraces}\text{]}$$

$$= \mathit{Scstraces}^{\mathtt{a}}_{\mathtt{cs}}(\mathit{cstraces}^{\mathtt{a}_{12}}(\mathtt{P}_1 \,[\![\, \mathtt{cs} \,]\!]\, \mathtt{P}_2), \mathit{cstraces}^{\mathtt{a}_3}(\mathtt{P}_3)) \quad \text{[definition of } \mathit{Scstraces}\text{]}$$

□

Theorem 4.

$$\mathit{selectS}^{\mathtt{a}}(\mathtt{P}) \subseteq \mathit{cstraces}^{\mathtt{a}}(\mathtt{P})$$

Proof. By induction.

Case: basic process. Trivial, because $\mathit{selectS}(\mathtt{P}) = \varnothing$.

Case: parallelism. Direct consequence of Theorem 2.

Case: choices. We can assume, without loss of generality, that

$$\mathtt{P}_1 = \mathtt{begin\ state}\ [\mathtt{x}_1 : \mathtt{T}_1] \bullet \mathtt{A}_1\ \mathtt{end} \text{ and } \mathtt{P}_2 = \mathtt{begin\ state}\ [\mathtt{x}_2 : \mathtt{T}_2] \bullet \mathtt{A}_2\ \mathtt{end}$$

with $\mathtt{x}_1 \cap \mathtt{x}_2 = \varnothing$, where $\mathtt{x}_1$ and $\mathtt{x}_2$ are the lists of state components of $\mathtt{P}_1$ and $\mathtt{P}_2$, and for simplicity, we sometimes use $\mathtt{x}_1$ and $\mathtt{x}_2$ as sets of variable names.

$\mathit{selectS}^{\mathtt{a}}(\mathtt{P}_1 \,\Box\, \mathtt{P}_2)$

$= \mathit{selectS}^{\mathtt{a}}(\mathtt{P}_1) \cup \mathit{selectS}^{\mathtt{a}}(\mathtt{P}_2)$ [definition of *selectS*]

$\subseteq \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1) \cup \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_2)$ [induction hypothesis]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \cup \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_2 \in \mathtt{T}_2, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2)$ [definition of *cstraces*]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2, (\mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2), \mathtt{A}_1) \cup \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2, (\mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2), \mathtt{A}_2)$

[definition of *cstraces*, and Propositions 1 and 2]

$$= \{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_2)\,\}$$
$$\cup$$
$$\{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_2) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\,\}$$

[definition of *cstraces*]

$= \{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid$ [property of sets]
$\quad \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \vee$
$\quad \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)$
$\quad \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\ \}$

$= \{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1 \,\Box\, \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)$
$\quad \bullet\ \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\ \}$

[Proposition 4]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2, (\mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2), \mathtt{A}_1 \,\Box\, \mathtt{A}_2)$ [definition of *cstraces*]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1 \,\Box\, \mathtt{P}_2)$ [definition of process external choice and *cstraces*]

The proof is similar for internal choice.

Case: sequence. We can assume, without loss of generality, that

$$\mathtt{P}_1 = \mathtt{begin\ state}\ [\mathtt{x}_1 : \mathtt{T}_1] \bullet \mathtt{A}_1\ \mathtt{end} \text{ and } \mathtt{P}_2 = \mathtt{begin\ state}\ [\mathtt{x}_2 : \mathtt{T}_2] \bullet \mathtt{A}_2\ \mathtt{end}$$

with $\mathtt{x}_1 \cap \mathtt{x}_2 = \varnothing$, where $\mathtt{x}_1$ and $\mathtt{x}_2$ are the lists of state components of $\mathtt{P}_1$ and $\mathtt{P}_2$, and for simplicity, we sometimes use $\mathtt{x}_1$ and $\mathtt{x}_2$ as sets of variable names.

$\mathit{selectS}^{\mathtt{a}}(\mathtt{P}_1\ ;\ \mathtt{P}_2)$

$= \mathit{selectS}^{\mathtt{a}}(\mathtt{P}_1) \cup \mathcal{N}^{\mathtt{a}}\ (\!|\ \mathit{tcstraces}^{\mathtt{a}_1}(\mathtt{P}_1)\ \mathrm{ccat}\ \mathit{selectS}^{\mathtt{a}_2}(\mathtt{P}_2)\ |\!)$ [definition of *selectS*]

$\subseteq \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1) \cup \mathcal{N}^{\mathtt{a}}\ (\!|\ \mathit{tcstraces}^{\mathtt{a}_1}(\mathtt{P}_1)\ \mathrm{ccat}\ \mathit{cstraces}^{\mathtt{a}_2}(\mathtt{P}_2)\ |\!)$ [induction hypothesis]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1) \cup$ [definitions of *cstraces* and *tcstraces*]
$\mathcal{N}^{\mathtt{a}}\ (\!|\ \mathit{tcstraces}^{\mathtt{a}_1}(\mathtt{w}_1 \in \mathtt{T}_1, \mathtt{x}_1 := \mathtt{w}_1, \mathtt{A}_1)\ \mathrm{ccat}\ \mathit{cstraces}^{\mathtt{a}_2}(\mathtt{w}_2 \in \mathtt{T}_2, \mathtt{x}_2 := \mathtt{w}_2, \mathtt{A}_2)\ |\!)$

$= \{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \mid \mathtt{x}_1 := \mathtt{w}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\ \} \cup$
$\{\ \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid$
$\quad \alpha\mathtt{st}_1 \leq \mathtt{a}_1 \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \mid \mathtt{x}_1 := \mathtt{w}_1 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \wedge$
$\quad \alpha\mathtt{st}_2 \leq \mathtt{a}_2 \wedge (\mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)$
$\quad \bullet\ \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}_1 \frown \mathtt{st}_2, \mathtt{c}_2 \wedge \mathtt{c}_3))\ \}$

[relational image, and definitions of ccat, *cstraces*, and *tcstraces*]

$= \{\, \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\ \} \cup$
$\{\ \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid$
$\quad \alpha\mathtt{st}_1 \leq \mathtt{a}_1 \wedge$
$\quad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{s}_2;\ \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{Skip}) \wedge$
$\quad \alpha\mathtt{st}_2 \leq \mathtt{a}_2 \wedge (\mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{s}_2;\ \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)$
$\quad \bullet\ \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}_1 \frown \mathtt{st}_2, \mathtt{c}_2 \wedge \mathtt{c}_3))\ \}$

[Propositions 1 and 2, $\mathtt{w}_2 \neq \mathtt{w}_1$, $\mathtt{x}_2 \notin FV(\mathtt{A}_1)$]
[by construction: $\mathtt{w}_2 \notin SV(\mathtt{s}_2, \mathtt{s}_3)$, $\mathtt{w}_2 \cap \alpha(\mathtt{st}, \mathtt{st}_1, \mathtt{st}_2) = \varnothing$]
[by construction: $FV(\mathtt{c}_2) \cap (\{\mathtt{w}_2\} \cup SV(\mathtt{s}_3)) = \varnothing$, $FV(\mathtt{c}_2) \cap \alpha\mathtt{st}_2 = \varnothing$]
[$FV(\mathtt{s}_2) \cap FV(\mathtt{A}_2) = \varnothing$, since $FV(\mathtt{s}_2) \subseteq FV(\mathtt{s}_1, \mathtt{A}_1)$]

$$
\begin{array}{ll}
= & \{\, st, c_3, s_3, A_3 \mid \alpha\mathtt{st} \le \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3) \,\} \\
& \cup \\
& \{\; \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \\
& \qquad \alpha\mathtt{st}_1 \le \mathtt{a}_1 \wedge \\
& \qquad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{s}_2;\ \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{Skip}) \wedge \\
& \qquad \alpha\mathtt{st}_2 \le \mathtt{a}_2 \wedge (\mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{s}_2;\ \mathtt{x}_2 := \mathtt{w}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \wedge \mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
& \quad \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}_1 \frown \mathtt{st}_2, \mathtt{c}_2 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \wedge \mathtt{c}_3)) \,\}
\end{array}
$$

[Proposition 3]

$$
\begin{array}{ll}
= & \{\, \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \le \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3) \,\} \\
& \cup \\
& \{\; \mathtt{st}, \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \\
& \qquad \mathtt{st} = \mathtt{st}_1 \frown \mathtt{st}_2 \wedge \alpha\mathtt{st}_1 \le \mathtt{a}_1 \wedge \alpha\mathtt{st}_2 \le \mathtt{a}_2 \wedge \\
& \qquad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \wedge \\
& \qquad (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
& \quad \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_3)) \,\}
\end{array}
$$

[property of sets and Proposition 3]

$$
\begin{array}{ll}
= & \{\; \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \le \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
& \quad \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_3)) \,\} \cup \\
& \{\; \mathtt{st}, \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \\
& \qquad \mathtt{st} = \mathtt{st}_1 \frown \mathtt{st}_2 \wedge \alpha\mathtt{st}_1 \le \mathtt{a}_1 \wedge \alpha\mathtt{st}_2 \le \mathtt{a}_2 \wedge \\
& \qquad (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \wedge \\
& \qquad (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
& \quad \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_3)) \,\}
\end{array}
$$

[$\mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_3)) = \mathcal{R}(\mathtt{st}, \mathtt{c}_3)$ because $\alpha\mathtt{st} \le \mathtt{a}$]

$$
\begin{array}{ll}
= & \{\; \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3, \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2 \mid \qquad \text{[property of sets]} \\
& \qquad (\alpha\mathtt{st} \le \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)) \\
& \qquad \vee \\
& \qquad \left(\begin{array}{l}
\mathtt{st} = \mathtt{st}_1 \frown \mathtt{st}_2 \wedge \alpha\mathtt{st}_1 \le \mathtt{a}_1 \wedge \alpha\mathtt{st}_2 \le \mathtt{a}_2 \wedge \\
(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \wedge \\
(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)
\end{array} \right) \\
& \quad \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_3)) \,\}
\end{array}
$$

$$
\begin{array}{ll}
= & \{\; \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3, \mathtt{st}_1, \mathtt{c}_2, \mathtt{s}_2, \mathtt{st}_2 \mid \alpha\mathtt{st} \le \mathtt{a} \wedge \\
& \qquad \left(\begin{array}{l}
(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
\vee \\
\left(\begin{array}{l}
\mathtt{st} = \mathtt{st}_1 \frown \mathtt{st}_2 \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1) \stackrel{\mathtt{st}_1}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip}) \wedge \\
(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \stackrel{\mathtt{st}_2}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)
\end{array} \right)
\end{array} \right) \\
& \quad \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3) \,\}
\end{array}
$$

[property of alphabets and $\mathcal{N}^{\mathtt{a}}(\mathtt{cst}) = \mathtt{cst}$ provided $\alpha\mathtt{cst} = \mathtt{a}$]

$$
\begin{array}{ll}
= \{\; \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \le \mathtt{a} \wedge (\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2 \mid \mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2 \models \mathtt{A}_1\ ;\ \mathtt{A}_2) \stackrel{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\
\quad \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3) \,\}
\end{array}
$$

[Proposition 5]

$= \mathit{cstraces}^{\mathtt{a}}(\mathtt{w}_1 \in \mathtt{T}_1 \wedge \mathtt{w}_2 \in \mathtt{T}_2, (\mathtt{x}_1, \mathtt{x}_2 := \mathtt{w}_1, \mathtt{w}_2), \mathtt{A}_1\ ;\ \mathtt{A}_2)$ [definition of *cstraces*]

$= \mathit{cstraces}^{\mathtt{a}}(P_1;\ P_2)$ [definition of process sequence and *cstraces*]

Case: hiding. We can assume, without loss of generality, that

$$\mathtt{P} = \mathtt{begin\ state}\ [\mathtt{x} : \mathtt{T}] \bullet \mathtt{A}\ \mathtt{end}$$

$$
\begin{array}{ll}
& \mathit{selectS}^{\mathtt{a}}(\mathtt{P}_1 \setminus \mathtt{cs}) \\
= & \mathcal{N}^{\mathtt{a}}\,(\!|\,(\!|\mathit{selectS}^{\mathtt{a}}(\mathtt{P})\,|\!)\restriction_{\mathtt{C}} \mathtt{cs}\,|\!) & \text{[definition of } \mathit{selectS}\text{]} \\
\subseteq & \mathcal{N}^{\mathtt{a}}\,(\!|\,(\!|\mathit{cstraces}^{\mathtt{a}}(\mathtt{P})\,|\!)\restriction_{\mathtt{C}} \mathtt{cs}\,|\!) & \text{[induction hypothesis]} \\
= & \mathcal{N}^{\mathtt{a}}\,(\!|\,(\!|\mathit{cstraces}^{\mathtt{a}}(\mathtt{w} \in \mathtt{T}, \mathtt{x} := \mathtt{w}, \mathtt{A})\,|\!)\restriction_{\mathtt{C}} \mathtt{cs}\,|\!) & \text{[definition of } \mathit{cstraces}\text{]} \\
= & \{\mathtt{cst} : \mathit{cstraces}^{\mathtt{a}}(\mathtt{w} \in \mathtt{T}, \mathtt{x} := \mathtt{w}, \mathtt{A}) \bullet \mathcal{N}^{\mathtt{a}}(\mathtt{cst}\restriction_{\mathtt{C}} \mathtt{cs})\} & \text{[relational image]} \\
= & \{\ \mathtt{st}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w} \in \mathtt{T} \mid \mathtt{x} := \mathtt{w} \models \mathtt{A}) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \bullet \mathcal{N}^{\mathtt{a}}(\mathcal{R}(\mathtt{st}, \mathtt{c}_2)\restriction_{\mathtt{C}} \mathtt{cs})\} \\
& & \text{[definition of } \mathit{cstraces}^{\mathtt{a}}\text{]} \\
= & \{\ \mathtt{st}, \mathtt{c}_3, \mathtt{s}_3, \mathtt{A}_3 \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{w} \in \mathtt{T} \mid \mathtt{x} := \mathtt{w} \models \mathtt{A} \setminus \mathtt{cs}) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \bullet \mathcal{R}(\mathtt{st}, \mathtt{c}_3)\} & \text{[Proposition 7]} \\
= & \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}_1 \setminus \mathtt{cs}) & \text{[definition of } \mathit{cstraces}^{\mathtt{a}}\text{]}
\end{array}
$$

□