Ana Cavalcanti · Marie-Claude Gaudel

Testing for Refinement in *Circus*

Received: date / Revised: date

Abstract *Circus* combines constructs to define complex data operations and interactions; it integrates Z and CSP, and, distinctively, it is a language for refinement that can describe programs as well as specification and design models. The semantics is based on Hoare and He's unifying theories of programming (UTP). Most importantly, *Circus* is representative of a class of refinement-oriented languages that combines facilities to specify abstract data types in a model-based style and patterns of interaction. What we present here is the *Circus* testing theory; this work is relevant as a foundation for sound test-generation techniques for a plethora of state-rich reactive languages. To cater for data operations, we define symbolic tests and exhaustive test sets. They are the basis for test-generation techniques that can combine coverage criteria for data and transition models. The notion of correctness is *Circus* refinement, a UTP-based generalisation of failures-divergences refinement that considers data modelling. Proof of exhaustivity exploits the correspondence between the operational and denotational semantics.

Keywords symbolic testing – specification-based testing – exhaustivity

1 Introduction

The use of formal models as a basis for testing techniques is well established; examples of early work can be found in [19,29,5,7,2,42]. On the other hand,

Ana Cavalcanti
University of York
Department of Computer Science
York YO10 5DD, England

Marie-Claude Gaudel
Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405;
CNRS, Orsay, F-91405, France

it was just recently that we presented a testing theory for CSP [58,38], a process algebra based on the notion of refinement.

In [14], we instantiate a well-established theory of formal testing [30] to CSP, using refinement as the notion of correctness. Here, we extend and generalise these results to a much richer process algebra: *Circus*. We tackle a problem that was not relevant in the context of CSP: testing based on models that cover both patterns of interaction and complex data structures specified in the predicative model-based style (of languages like Z, VDM, and B).

We propose and justify the use of novel symbolic representations of traces, initials, acceptance sets, and tests. What we achieve is the identification and formalisation of a theoretical framework that (1) provides a clear route for arguably sound symbolic techniques for test-case generation; and (2) applies to refinement notations suited for state-rich reactive systems, with complex data types and operations specified in a predicate-based style.

Circus, in particular, combines CSP [58], Z [73], and a refinement theory and technique [17]. It has a semantics based on Hoare and He's unifying theories of programming (UTP) [39]. It is a relational model rendered in a predicative style like that of Z; it supports combinations of programming constructs from different notations and paradigms. It caters for the CSP-based constructs of *Circus*, but also, unlike the CSP failures-divergences model, it caters for the specification and refinement of data operations described in an assertional style, and the use of miraculous programs in refinement [48].

There are several combinations of a process algebra with a state-based formalism; examples are discussed in [1,28,65,64,23,62,68,10,24,45,40,51, 41]. *Circus* distinguishes itself as a language for refinement, that supports modelling of high-level specifications, designs, and concrete programs. It is representative of a class of languages that provide facilities to model data types, using a predicate-based notation, and patterns of interactions, without imposing architectural restrictions. It is this feature that makes it suitable for reasoning about both abstract and low-level designs. *Circus* has been applied in areas like control systems [53,13], mobility [67,66], real-time applications [61], and hardware languages [11]; tools are under development [26, 71]. It is important to emphasise, however, that our results are relevant in the context of all the works cited above, where a language that integrates model-based data specifications and reactive constructs is considered.

The use of *Circus* as a basis for our study benefits from the fact that it has both a denotational and operational semantics, with a justified connection. The UTP provides a well established relational framework for justifying the soundness of our results. In addition, *Circus* has a well understood simple account of the central notion of refinement.

Interpreting a testing experiment in *Circus*, as in any formal framework is not straightforward, because we test a system, not a model. To bridge the gap between them, we use testability hypotheses in [2]. These are assumptions on the system under test (SUT) that are essential to the proof that the success of a testing campaign entails correctness. Complementary tests or proofs are typically necessary to establish that the testability hypotheses are valid.

In our *Circus* theory of testing, the first testability hypothesis is that the SUT behaves like some unknown *Circus* process SUT_{Circus}. This means that, in any environment, execution of the SUT or execution of SUT_{Circus} gives the same set of observations. In this context, even though the SUT is not a *Circus* process, we can use refinement to compare it to a given *Circus* specification. This requires, however, that events used in the specification are perceived as atomic and of irrelevant duration in the SUT.

This kind of testability hypothesis has been widely used in formalisations of testing experiments. As early as in the 70s, Chow [19] raised this issue for Finite State Machine (FSM) descriptions. He assumes that the SUT behaves as an FSM, and all subsequent FSM-based techniques rely on this testability hypothesis [42]. Similarly, methods for IO-Transition Systems assume that the SUT behaves as an IO-automaton [69], and methods for algebraic specifications assume that the SUT behaves as a many-sorted algebra [2].

Our second testability hypothesis is that the SUT satisfies a complete testing assumption: when a test experiment is performed a number of times, then all possible (nondeterministic) behaviours of the SUT are observed. The definition of the required number of repetitions of the experiments is usually based on knowledge of the SUT, like, for example, the level of internal parallelism that may lead to nondeterminism, or the way choices are resolved. It is also possible to use knowledge of the execution environment or statistical techniques. The complete testing assumption on the SUT is standard in the testing literature and practice. Basically, there is no way of drawing conclusions about testing experiments involving a nondeterministic SUT without such an assumption [7,27]. This, however, imposes no restrictions on the specifications, which, in the case of *Circus*, in particular, are typically nondeterministic, and even possibly unboundedly nondeterministic.

In our formalisation of the observations (traces, initials, and acceptances) that can be done when performing test experiments derived from a *Circus* specification, we take advantage of the symbolic nature of the *Circus* operational semantics. In contrast to the CSP semantics, the operational semantics for *Circus* [72] gives a symbolic account of the evolution of a system. This caters for the rich data constructs; from the point of view of testing, it induces a natural symbolic characterisation of tests and test sets.

Our main contribution is a symbolic characterisation of an exhaustive test set. This is a suite of symbolic tests such that, if we were to run experiments using all the concrete tests obtained by instantiating these symbolic tests, getting the right observations is equivalent to establishing that the SUT is correct with respect to the given *Circus* specification and in accordance to refinement. We, of course, do not propose that a practical testing technique uses all the tests in these sets, since they are typically infinite or very large. Their definition, however, is a fundamental basis to justify pragmatic selection and test-case generation techniques. The set of tests that we propose that a practical technique selects and generates is a subset of the exhaustive test set that we present here. We establish exhaustivity in terms of the usual operational concepts, but we have a clear link to the denotational semantics. This simplifies the proofs, and allows for a formal algebraic style. Our contribution in this paper is mainly of a theoretical nature, but it is a foundation for practical techniques that we are currently exploring.

In our previous work on CSP, we have provided only concrete tests; that was appropriate, since the data language of CSP is rather simple when compared to that of *Circus*. In any case, we could have adopted a similar approach for *Circus*, and our results transfer with almost no change. In the context of *Circus*, however, these concrete tests make it more difficult to exploit the exhaustive test set effectively, because of the rich data language of *Circus*. For any non-trivial specification, the exhaustive test set is infinite, and test selection and test-data generation techniques are essential in practice. Our symbolic tests are ideal bases to consider well known strategies using constraints decomposition and solving [2,21,37] to explore the rich data models, and ensure meaningful coverage of the observations and of the specification. In addition, the symbolic tests cater for the rich, potentially infinite, data types of *Circus* models, with operations specified in the Z predicative style, as opposed to the CSP types, which are specified using a functional language.

In summary, our symbolic tests and the ingredients used to define them, namely, constrained symbolic traces, initials, and acceptance sets, are a prerequisite for proposing and justifying test-data generation techniques in any language combining control and complex data types. They, for instance, identify the relevant constraint-solving problems involved. Our proofs of exhaustiveness confirm their suitability in the context of testing techniques for *Circus* refinement. The approach, however, is of more general interest.

The notion of refinement in *Circus* is a natural generalisation of failures-divergences refinement induced by the UTP model. It considers information about interactions, like in the canonical failures-divergences model of CSP, but also about data and associated operations. In [16], we calculate characterisations in *Circus* of traces refinement, which, like in CSP, amounts to traces inclusion, and also of the relation usually called *conf* (for conformance), which requires reduction of deadlock. We also prove in [16] that the combination (conjunction) of traces refinement and *conf* corresponds to process refinement in *Circus*, for divergence-free processes. Accordingly, our characterisation of an exhaustive test set is the union of an exhaustive test set for traces refinement, and another for *conf*.

We work with divergence-free specifications, since we take the view that divergence in a specification is a mistake. In addition, since it is not possible to distinguish a divergent from a deadlocked system using testing, if the SUT diverges, this is observed as a deadlock and reported as such in the verdict.

A foundational work on testing theory for process algebras, more specifically CCS, is that of de Nicola and Hennessy [49]. As most works on testing based on process descriptions, the theory considered here was influenced by [49] in, for example, its definition of a test execution. The work in [49] focus on testing equivalences, but in [33] Hennessy covers refinement testing with respect to his acceptance-tree semantics. May and must-tests are introduced to characterise trace refinement and failures refinement. The framework presented is, however, different from ours, in that two models are compared. We are interested in comparing a specification model and a system (under test). To justify our results, we assume, as explained previously, that there is an unknown model of the system, but we have an extremely limited knowledge of it. We observe that it is because de Nicola and Hennessy work with models that the results in [49] do not require testability hypotheses.

In the next section, we give an overview of *Circus*: we use an example to introduce the notation, and describe its operational semantics. To define symbolic tests, we need to define symbolic characterisations of traces, initials, and acceptances for *Circus*; this is the subject of Section 3. In Section 4, we present our results for traces refinement, and in Section 5 our results for *conf*. Finally, in Section 7, we discuss related and future work. Appendix A provides more information about the operational semantics, covering transition rules for external choice and parallelism, and a few examples. The proofs of the lemmas that are omitted in Sections 4 and 5 can be found in [15].

2 *Circus* and its operational semantics

To explain the *Circus* notation, we provide as an example the model of a cash machine that keeps track of the status of cards, pin numbers, and the bank of notes available. We also use this example to illustrate our symbolic characterisations of tests, and to illustrate, in Appendix A, the use of the definitions that specify the *Circus* operational semantics. The example covers the main features of *Circus* relevant to our work: state, combination of nondeterministic data operations and patterns of interaction, and concurrency. It is an abstract model, but, as we explain later on, it can be used as a basis for testing concrete implementations. A complete description of *Circus*, including examples, and its refinement technique can be found [52,54].

2.1 Notation

A *Circus* model is a sequence of paragraphs, just like in Z, but they can also declare channels and processes. Figure 1 gives an overview of the structure of our example; we observe that the definition of the *CashMachine* process is also a sequence of paragraphs. First of all, however, we have a paragraph that declares the sets *CARD* and *PIN* of valid cards and pin numbers. We use the Z notation for introducing given sets.

$[CARD, PIN]$

We next declare some channels. Requests for money are accepted by the cash machine through the channel *incard*, which takes a card, a pin number and an amount to be withdrawn: the inputs are triples.

channel $incard : CARD \times PIN \times \mathbb{N}_1$

The amount is a positive natural number. Cards are returned through a channel *outcard*, unless there is a problem with the card and it is retained.

channel $outcard : CARD$

The notes kept in and dispensed by the cash machine are those whose denominations are in the set *Note* defined below (in the standard Z notation).

$Note == \{10, 20, 50\}$

For simplicity, we consider just a few notes, and do not address the fact that the amount requested must be decomposable in terms of the notes available. If it is not, the machine fails to dispense the cash. In our model, cash is represented as a bag of notes: elements of the set *Cash*.

$Cash == \mathrm{bag}\, Note$

If there is enough money in the machine and a way of providing the requested amount, the cash is output through a channel *cash*.

$\mathbf{channel}\ cash : Cash$

The cash machine has two main components: a card verifier, which accepts requests and decides whether the card should be returned and the cash dispensed, and a cash controller, which dispenses the cash if possible, and refills the note bank. These components interact through the channels below.

$\mathbf{channel}\ disp : \mathbb{N}_1;\ ok$

The channel *disp* is used by the card verifier to tell the cash controller to dispense a given amount, and *ok* is used by the cash controller to tell the card verifier that it has concluded its operation. The channel *ok* does not have a type; it is not used to communicate values, but just for synchronisation. This is also the case of the channel *refill* defined below.

$\mathbf{channel}\ refill$

This channel is used to accept requests to refill the machine

A *Circus* process models a system or a component. Just like in CSP, it interacts with its environment and other processes via channels. In *Circus*, however, a process encapsulates a state defined just like in Z. Our model defines the process *CashMachine*; its only state component is a function *noteBank* that records, for each denomination, the amount of notes available.

$\mathbf{process}\ CashMachine \mathrel{\widehat=} \mathbf{begin}$

The state is defined by a schema, namely *CMState*, which declares *noteBank* as a total function. In this example, we do not have an elaborate state invariant (which is restricted to the functional property of *noteBank*).

$\mathbf{state}\ CMState == [\, noteBank : Note \to \mathbb{N}\,]$

The function *pin* defines the pin number of each valid card. It is declared using a Z axiomatic description, but its scope is restricted to the process.

$|\ \ pin : CARD \to PIN$

For simplicity, we assume that the pin numbers are constant.

$[CARD, PIN]$
channel $incard : CARD \times PIN \times \mathbb{N}_1$
channel $outcard : CARD$
$Note == \{10, 20, 50\}$
$Cash == \mathrm{bag}\, Note$
channel $cash : Cash$
channel $disp : \mathbb{N}_1;\ ok$
channel $refill$

process $CashMachine \mathrel{\widehat=}$ **begin**

state $CMState == [\, noteBank : Note \to \mathbb{N}\,]$

$|\quad pin : CARD \to PIN$

$CardV \mathrel{\widehat=} \ldots$

$$\begin{array}{|l}
\multicolumn{1}{l}{DispenseNotes} \\
\Delta CMState \\
a? : \mathbb{N};\ notes! : Cash \\
\hline
\ldots \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{DispenseError} \\
\Xi CMState \\
a? : \mathbb{N};\ notes! : Cash \\
\hline
\ldots \\
\hline
\end{array}$$

$Dispense == DispenseNotes \lor DispenseError$

$|\quad cap : \mathbb{N}$

$CashC \mathrel{\widehat=} \ldots$

$\bullet\ (CardV \ [\![\ \{\} \mid \{\!|\ disp, ok\ |\!\} \mid \{noteBank\}\ ]\!]\ CashC) \setminus \{\!|\ disp, ok\ |\!\}$

end

Fig. 1 Sketch of the cash machine model

The card verifier is defined below using basically CSP notation. It accepts a request $incard?c.(pin\ c)?a$; this is an input of any card c, the particular pin number $pin\, c$, and any amount a. It then decides whether to retain the card, output it using *outcard* and no money, or ask the cash controller to dispense the requested amount. The decision is nondeterministic; it is defined by factors outside of this model: status of the card, balance on the corresponding account, and so on. If the card verifier asks for the cash to be dispensed, then it waits for an *ok* from the controller to indicate that it is finished (and the verifier can proceed recursively to accept a new request).

$$CardV \mathrel{\widehat=} \left(\begin{array}{l} incard?c.(pin\ c)?a \rightarrow \\ \quad \left(\begin{array}{l} CardV \\ \sqcap \\ outcard!c \rightarrow CardV \\ \sqcap \\ disp!a \rightarrow ok \rightarrow outcard!c \rightarrow CardV \end{array}\right) \end{array}\right)$$

Nondeterminism here is in the pattern of interaction, and it is explicitly indicated using the CSP construct $\sqcap$ for internal choice.

The *DispenseNotes* data operation takes an amount $a?$ as input and produces a bag of notes $notes!$ as output; it also updates *noteBank*. It is defined using a Z schema that specifies a relation on *CMState*. (This is the standard Oxford style used in Z to define data operations.) With the declaration $\Delta CMState$, we introduce the variables *noteBank*, to represent the value of the state component before the operation, and $noteBank'$, to represent its value after the operation. The definition of the schema $\Delta CMState$ is implicitly established by the Z (and the *Circus*) semantics.

$$\begin{array}{|l} \textit{DispenseNotes} \\ \hline \Delta CMState \\ a? : \mathbb{N};\ notes! : Cash \\ \hline \Sigma\, notes! = a? \\ \left(\begin{array}{l} \forall n : Note \bullet \\ \quad \left(\begin{array}{l} (notes! \sharp n) \leq noteBank\ n \land \\ noteBank'\ n = (noteBank\ n) - (notes! \sharp n) \end{array}\right) \end{array}\right) \\ \hline \end{array}$$

The value of $notes!$ is nondeterministically chosen: it is any bag $notes!$ whose sum $\Sigma\, notes!$ of its elements is equal to $a?$, and such that, for each note denomination n, the number $notes! \sharp n$ of occurrences of n is less than or equal to the number $noteBank\ n$ of notes of denomination n in the bank.

If there is no such bag, we have an error: the output is the empty bag $[\![\ ]\!]$, and the state is not changed. This is defined by the schema *DispenseError* below; following the Z practice, it includes the schema called $\Xi CMState$ to declare implicitly the variables *noteBank* and $noteBank'$ representing the state components and define (also implicitly) that their values are equal.

$$\begin{array}{|l} \textit{DispenseError} \\ \hline \Xi CMState \\ a? : \mathbb{N};\ notes! : Cash \\ \hline \neg\ \exists ns : Cash \bullet \Sigma\, ns = a? \land \forall n : Note \bullet (ns \sharp n) \leq noteBank\ n \\ notes! = [\![\ ]\!] \\ \hline \end{array}$$

The total operation to *Dispense* cash is the schema disjunction of the operations *DispenseNotes* and *DispenseError*.

$$Dispense == DispenseNotes \lor DispenseError$$

For conciseness, we omit the definition of the operation Σ for bags.

The cash controller *CashC* offers the choice to *refill* the bank or *disp*ense some money. For simplicity, we assume that when the machine is refilled, it then has a number *cap* of notes of each denomination.

$$\mid \quad cap : \mathbb{N}$$

This is a constant that reflects the capacity of the cash machine.

In the definition of *CashC*, we use an assignment to *noteBank*, instead of a data operation defined by a Z schema, to define the value of the state after a synchronisation on *refill*. This illustrates the possibility of use of programming constructs as well as abstract specifications in *Circus*. (In particular, it is possible to define an executable *Circus* model.)

If the cash controller receives a request *disp?a* to dispense an amount *a* of cash, it uses the operation *Dispense* defined previously to determine how cash is to be dispensed. To use that operation, *CashC* declares a local variable *notes*. The input variable *a* and the local variable *notes* now in scope are associated with the input and output of *Dispense*, which then assigns an appropriate value to *notes*. If that value is not the empty bag, then the cash is dispensed using the channel *cash*, and then the message *ok* is sent (to *CardV*). If, on the other hand, the bag is empty, then it is not possible to output the amount of cash requested and the *ok* message is sent directly.

$$CashC \mathrel{\widehat=} \left(\begin{array}{l} refill \rightarrow (noteBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\} \;;\; CashC) \\ \Box \\ disp?a \rightarrow \left(\begin{array}{l} \mathbf{var}\ notes : Cash \bullet \\ \left(\begin{array}{l} Dispense; \\ \left(\begin{array}{l} (notes \neq [\![\ ]\!]) \mathrel{\&} cash!notes \rightarrow ok \rightarrow CashC \\ \Box \\ (notes = [\![\ ]\!]) \mathrel{\&} ok \rightarrow CashC \end{array}\right) \end{array}\right) \end{array}\right) \end{array}\right)$$

This illustrates the free combination of specification and programming constructs, and the free combination of data operations and communications. There is no direct association between interactions and state changes.

So far, we have defined just components of (the model of) *CashMachine*. In particular, the schemas *DispenseNotes*, *DispenseError*, and *Dispense*, which are data operations, and *CardV* and *CashC*, which are actions of *CashMachine*. As we have seen, they have access to the state of the process, and are defined using a combination of Z, CSP, and guarded commands. They are used in the specification below of a main (nameless) action that defines the behaviour of *CashMachine*. This is a parallel composition of the *CardV* and *CashC* components, synchronising on the channels *disp* and *ok*.

In *Circus*, to avoid conflicts in the access to variables, a parallel composition of actions defines the disjoint sets of variables to which each of the parallel actions have write access. All the actions can read the value of all the variables before the parallelism starts, but can modify only the variables in their associated sets. In our example, *CardV* does not update the state, and so is associated with the empty set $\{\,\}$ of variables. On the other hand, *CashC* updates *noteBank*, and so it is associated with $\{noteBank\}$.

The parallel composition also defines the channels on which communication requires interaction from both parallel actions. In our example, they are *disp* and *ok*. This means, for example, that *CardV* can engage on communications using *incard* and *outcard* independently from *CashC*. This freedom is an extra source of nondeterminism in the specification model.

The channels *disp* and *ok* are used only for communication between the internal components *CardV* and *CashC* of *CashMachine*. Such communications are of no interest to the user of a cash machine, and so they are hidden. Just like in CSP, communications on hidden channels are not visible.

To conclude, the main action of *CashMachine* is as follows; it is separated from the previously defined (auxiliary) actions by a $\bullet$. The parallelism of actions A_1 and A_2 is written $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, where ns_1 and ns_2 are the sets of the names of the variables to which A_1 and A_2 are associated in the parallelism, and cs is the set of channels on which synchronisation between A_1 and A_2 is required. Additionally, $A \setminus cs$ is the action obtained by hiding the (communications on the) channels in set cs in the action A.

$$\bullet\ (CardV \llbracket \{\} \mid \{\!| disp, ok |\!\} \mid \{noteBank\} \rrbracket\ CashC) \setminus \{\!| disp, ok |\!\}$$

end

Since *CardV* and *CashC* synchronise on (just) the channels *disp* and *ok*, we have that the cash machine can be refilled while a request for cash is being processed, but not when cash is actually being dispensed.

As highlighted in Figure 1, the main action is the last component of an explicit process definition, like that above. The **end** marks the end of the definition, and matches the **begin** at the start of the process declaration.

Interaction with *CashMachine* is only possible via the channels *incard*, *outcard*, *refill*, and *cash*, in the way defined by its main action above. There is no possibility of direct access to its state, which is encapsulated. In *Circus*, it is also possible to combine basic processes, which are defined as above using Z and CSP constructs; the combinators are the usual CSP operators for internal and external choice, parallelism, interleaving and so on.

2.2 Operational semantics

In [72], a Plotkin-style operational semantics for *Circus* is presented. In this section, we summarise the main definitions for processes and actions. The transition relation is defined in the context of the UTP: it is characterised in the UTP theory used to give a denotational semantics to *Circus*, so that the rules of the operational semantics discussed below are laws of this theory.

The predicates involved in the configurations of the transition relation, in particular, are (texts of) predicates of the UTP general relational theory. The treatment of alphabets is that adopted in the UTP as well. In addition, we observe that the general UTP theory of relations is a complete lattice, so that the greatest lower bound and least upper bound operators are available (as options to provide and account of recursion, for instance.)

$$
\begin{array}{lll}
\mathsf{Process} & ::= & \textbf{begin}\ [\textbf{state}\ \mathsf{Schema\text{-}Exp}]\ \mathsf{PParagraph}^{*} \bullet \mathsf{Action}\ \textbf{end} \\
 & | & \mathsf{Process};\ \mathsf{Process} \\
 & | & \mathsf{Process} \mathrel{\Box} \mathsf{Process} \\
 & | & \mathsf{Process} \sqcap \mathsf{Process} \\
 & \ldots & \\
 & | & \textbf{begin} \\
 & & \quad [\textbf{state}\ \mathsf{Schema\text{-}Exp}]\ \mathsf{loc}\ (\mathsf{Predicate} \mid \mathsf{Predicate})\ \bullet \mathsf{Action} \\
 & & \textbf{end}
\end{array}
$$

Fig. 2 Sketch of the BNF definition of Processes

2.2.1 Processes

The labelled transition system that specifies the operational semantics of a *Circus* process is defined by a transition relation between texts of processes. These belong to the syntactic category Process, defined in the *Circus* BNF [54]; Figure 2 briefly reproduces a sketch the specification of Process. (The omitted productions just define the complete list of CSP operators that can be used to combine *Circus* processes, including parallelism and hiding.)

A basic process specification **begin state** $[x : T] \bullet A$ **end** declares a schema $[x : T]$ that defines the process state, and its main action A. In definitions, we consider a state defined by a single schema with a single component x of type T, but any schema expression (element of Schema-Exp) can be used, and also the state can be omitted altogether. The definitions that cater for this generality are long, but straightforward.

As exemplified previously, a basic process definition includes a number of paragraphs that define local auxiliary actions used in the definition of the main process action. In the *CashMachine* example, we have, for instance, paragraphs that define the actions *CardV* and *CashC* used directly to define the main action. Such paragraphs are called process paragraphs; their syntax is captured by the category PParagraph, and is based on that of Z, CSP, and the refinement calculus. In definitions, we assume that there are no local actions. In this case, the main action A is a self-contained definition of the behaviour of the process. Any basic process can be rewritten (using syntactic transformations) in this form. In the specification of *CashMachine*, for instance, we could have avoided the definition of the actions *Dispense*, *CashC*, and so on, and inlined all the specifications in the main action. Of course, this is not the style of specification encouraged in *Circus*, but from the point of view of semantic definitions, there is no loss of generality.

For the operational semantics, the definition of Process is extended to add an extra form of basic process that includes an additional clause loc that records the (local) value of the state, as the execution of the process evolves. The state is represented by (the text of) two predicates. The first constrains the values of symbolic variables used to represent the values of the state components (and any extra variables in scope). The second predicate relates the state components to the symbolic variables. In simple examples, we write this predicate as an assignment; we adopt the view of the UTP that programs are predicates. Like in the UTP, we use the typewriter font to distinguish predicate texts from the predicates that they denote.

Below, we give the rules that define the transition relation $\mathtt{p}_1 \xrightarrow{\mathtt{l}} \mathtt{p}_2$ for *Circus* processes; it associates processes $\mathtt{p}_1$ and $\mathtt{p}_2$, and a label $\mathtt{l}$. When $\mathtt{p}_1 \xrightarrow{\mathtt{l}} \mathtt{p}_2$ is in the transition relation, we say that there is a transition from $\mathtt{p}_1$ to $\mathtt{p}_2$ with label $\mathtt{l}$. Intuitively, if the label is ϵ, this means that the execution of the process p_1 can evolve silently (without any interaction with the environment), so that we now 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 transition rule (1) below defines that the first step in the execution of a basic process is a silent transition that introduces the loc clause. In the target process, the value of the state component x is represented by a fresh symbolic variable w_0 constrained to have a value of the type T of x. The constraint in the clause loc is also on the symbolic variables used in the labels. In all rules, the symbolic variables introduced are assumed to be fresh.

$$\frac{}{\begin{pmatrix}\mathtt{begin}\\ \quad\mathtt{state}\ [\mathtt{x} : \mathtt{T}]\\ \quad\bullet\ \mathtt{A}\\ \mathtt{end}\end{pmatrix} \xrightarrow{\epsilon} \begin{pmatrix}\mathtt{begin}\\ \quad\mathtt{state}\ [\mathtt{x} : \mathtt{T}] \mid \mathtt{loc}\ (\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{x} := \mathtt{w}_0)\\ \quad\bullet\ \mathtt{A}\\ \mathtt{end}\end{pmatrix}} \tag{1}$$

For *CashMachine*, the first step of execution introduces a loc clause that contains a predicate that states that the type of a fresh symbolic variable w_0 is the same as that of *noteBank*, and the assignment $\mathtt{noteBank} := \mathtt{w}_0$.

For a basic process that already contains a clause loc, the transition rule (2) below applies. Evolution is determined by the transition relation $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ for actions presented in the next section.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}{\begin{pmatrix}\mathtt{begin}\\ \quad\mathtt{state}\ [\mathtt{x} : \mathtt{T}] \mid \mathtt{loc}\ (\mathtt{c}_1 \mid \mathtt{s}_1)\\ \quad\bullet\ \mathtt{A}_1\\ \mathtt{end}\end{pmatrix} \xrightarrow{\mathtt{l}} \begin{pmatrix}\mathtt{begin}\\ \quad\mathtt{state}\ [\mathtt{x} : \mathtt{T}] \mid \mathtt{loc}\ (\mathtt{c}_2 \mid \mathtt{s}_2)\\ \quad\bullet\ \mathtt{A}_2\\ \mathtt{end}\end{pmatrix}} \tag{2}$$

For processes defined in terms of other processes, there is no need to give transition rules. Their semantics is defined by transformations that remove the process operators, by combining the bodies of the basic processes and their main actions accordingly [54]. For example, an external choice $p_1 \Box p_2$ of basic processes p_1 and p_2 can be written as a single basic process whose state includes the components of the states of p_1 and p_2, and whose main action is obtained by using the action (as opposed to process) external choice operator to combine the main actions of p_1 and p_2.

2.2.2 Actions

The configurations of the transition system for the semantics of *Circus* actions are triples $(\mathtt{c} \mid \mathtt{s} \models \mathtt{A})$ where $\mathtt{c}$ is a constraint over the symbolic variables in use, $\mathtt{s}$ an assignment of values to the all the variables in scope, and $\mathtt{A}$ a

Circus action. We present below the transition rules for a selected subset of the *Circus* actions: assignments, schemas, output and input prefixing; in the Appendix A, we cover internal and external choice, and parallelism.

The rule (3) for an assignment $\mathtt{v} := \mathtt{e}$ defines that its execution is silent, changes the current constraint $\mathtt{c}$ on symbolic variables by recording that the fresh variable w_0 has value e, changes the current state assignment $\mathtt{s}$ by associating w_0 to v, and then terminates. The definitions of the new constraint and state use the UTP sequence operator on predicates, which corresponds to relational composition, and whose definition we give below.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{v} := \mathtt{e}) \xrightarrow{\epsilon} (\mathtt{c} \land (\mathtt{s};\ \mathtt{w_0} = \mathtt{e}) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w_0} \models \mathtt{Skip})} \qquad (3)$$

In the UTP, predicates are alphabetised: the alphabet αp of a predicate p is the set of variables that represent observations of interest, and that are possibly restricted by p. In our case, these are the variables in scope: the state components and any extra variables that have been declared later. For predicates that represent relations, the alphabet may include dashed variables v' and, just like in Z, these predicates relate two observations of the variables. In this case, we can partition the alphabet into input (undecorated) variables $in\alpha p$, and output (dashed) variables $out\alpha p$. If, for predicates p_1 and p_2, we have that by undashing the variables in the output alphabet of p_1, we obtain the set of input variables of p_2, then we can compose these predicates in sequence. The predicate $p_1;\ p_2$ is defined as $p_1;\ p_2 \ \widehat{=}\ \exists\, v_0 \bullet p_1[v_0/v'] \land p_2[v_0/v]$. The intermediate state characterised by the final state of p_1 and the initial state of p_2 is represented using 0-subscripted variables, and quantified (hidden).

Going back to the transition rule (3), we observe that the predicate $w_0 = e$ has in its alphabet only the state components (possibly used in e). The corresponding dashed variables are not in the alphabet; in other words, this is a condition: a special case of a relational predicate in which there is no output variables. It is important to observe that w_0 is a (fresh) symbolic variable, and so not part of the alphabet of the constraint (and not related to the 0-subscripted variables used above to define a sequence $p_1;\ p_2$). So, the constraint $\mathtt{s};\ \mathtt{w_0} = \mathtt{e}$ restricts the value of w_0 to that of e when evaluated in the state characterised by the assignment $\mathtt{s}$. For example, if the state contains two variables x and y, and $\mathtt{s}$ is $\mathtt{x}, \mathtt{y} := \mathtt{w_1}, \mathtt{w_2}$, then $(x, y := w_1, w_2);\ (w_3 = x + y)$ is equivalent to $w_3 = w_1 + w_2$. In addition, as expected, the predicate $s;\ v := w_0$ defines that the value of v is w_0, and those of all other variables are as in s. In the UTP, an assignment $x := e$ with alphabet $\{x, y, \ldots, z, x', y', \ldots, z'\}$ is defined as $x := e \ \widehat{=}\ x' = e \land y' = y \land \cdots \land z' = z$.

An important observation is that the constraint c of the assignment configuration is a hypothesis in the transition rule (3). This ensures that transitions only occur when the constraint is satisfiable: configurations with unsatisfiable constraints are stuck, that is, there are no valid transitions from such configurations. As further illustrated below, all rules enforce such property, and those that strengthen the constraints ensure that they are still satisfiable. Therefore, if we start from a satisfiable constraint, the transitions lead only to configurations with a satisfiable constraint. In the case of the assignment rule, the new constraint is necessarily satisfiable whenever c is, because, as already explained, the extra conjunct added to the constraint only equates the fresh constant w_0 to the value of the total expression e.

The rule (4) for an action defined by a schema $\mathtt{Op}$ that specifies an operation is similar to that for an assignment. The difference is that now the assignment is to all variables v in the output alphabet of the current state assignment $\mathtt{s}$. For simplicity, the set v is used in a (potentially multiple) assignment to denote a list of all its variables.

$$\frac{c \wedge (s;\ \mathbf{pre}\ Op)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{Op}) \xrightarrow{\epsilon} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{Op}[\mathtt{w_0}/\mathtt{v}']) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w_0} \models \mathbf{Skip})} \quad v = out\alpha s \tag{4}$$

The values of the fresh constants w_0 are determined in the constraint by the restrictions imposed by Op on v'. Both v and w_0 are being used to denote lists of variables and fresh constants. To make sure that the restrictions on v', and therefore the new constraint, are satisfiable, the hypothesis requires that the constraint c of the schema configuration and the precondition of Op (as calculated using the standard Z **pre** Op operator) in the current state s are satisfiable. We omit the rule for when the precondition does not hold: the execution of Op aborts, a behaviour captured by the action *Chaos* in *Circus*.

Communications and synchronisations are described using prefixing actions. We consider separately output prefixings $d!e \to A$, where the value of the expression e is output through the channel d before the action behaves like A, and input prefixings $d?x : T \to A$, where an input of the type T of d is accepted through d and assigned to the local variable x. As illustrated in the previous section, inputs are not (necessarily) decorated by their types, but it is assumed that this type is available, so that the local variable x can be declared appropriately; type checking provides this information.

The transitions from a prefixing are labelled with an input $\mathtt{d?w_0}$ or an output $\mathtt{d!w_0}$ communication. It is necessary to distinguish whether the communication represents an input or an output because an input $\mathtt{d?w_0}$ records that every event $d.v$, where the value v satisfies the restrictions on w_0 in the target configuration, is accepted. On the other hand, an output $\mathtt{d!w_0}$ indicates that there is only a single such event $d.v$ that is accepted.

For example, in the *CashC* action of the cash machine, we have an input $disp?a : \mathbb{N}$; so, *CashC* is prepared to engage on any of the events $disp.0$, $disp.1$, and so on. The synchronisation defines the value of the variable a, and any value of the right type is acceptable. On the other hand, in the output $cash!notes$, the value of $notes$ is nondeterministically chosen by *Dispense*. Any bag of notes that satisfies the specification *Dispense* may be chosen for output, but *CashC* is prepared to accept a synchronisation only on the single event in which that unique (although loosely defined) value is communicated.

The transition rule (5) for an output prefixing is presented below.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d!e} \to \mathtt{A}) \xrightarrow{\mathtt{d!w_0}} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{w_0} = \mathtt{e}) \mid \mathtt{s} \models \mathtt{A})} \tag{5}$$

The value of w_0, used to represent the communicated output, is recorded to be that of e. We also treat as an output a prefixing involving a synchronisation channel, like *ok* in our example, which does not communicate any values.

In the rule (6) for input prefixing, w_0 represents the value input. Its value is restricted only by the type T of the channel, and the hypothesis requires that T is non-empty, so that $w_0 \in T$ is satisfiable.

$$\frac{c \wedge T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \rightarrow \mathtt{A}) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\ x} := \mathtt{w_0} \models \mathtt{let\,x} \bullet \mathtt{A})} \qquad (6)$$

In this rule, it is also necessary to declare the fresh input variable x. In the state assignment, the UTP **var** construct, which introduces a fresh variable (but does not close its scope) is used. In fact, for convenience, a new construct **var** $v := e$ is introduced; it is defined as the sequence **var** v; $v := e$ that declares the fresh v and assigns e to it.

Finally, the **let** action in the target configuration of the transition rule (6) is not part of the standard *Circus* syntax. We introduce this construct to facilitate the definition of the operational semantics; **let** $x \bullet A$ records the fact that the variable x used in A is local. We omit the transition rules for it, but in a few words, when `A` is reduced to `Skip`, then the `let` is eliminated, and the scope of x is closed in the state assignment.

The transition rules for internal and external choice, and parallelism are discussed in Appendix A, where we also give examples of the use of the rules, including those presented above. The whole set of rules can be found in [72]. In the next section, we use the labelled transition system that they define to specify symbolic characterisations of traces, initials, and acceptances.

3 Symbolic traces, initials, and acceptances

To establish correctness with respect to traces refinement, we need to test the SUT against the traces that are not traces of the reference *Circus* specification SP, and check that they are refused. In fact, it is enough to consider the minimal prefixes of forbidden traces that are forbidden themselves: if a trace $\langle a, b, a \rangle$ is forbidden, and we confirm that it is refused by the SUT, we no longer need to check any of its extensions; if it is accepted, an error is already identified. We, therefore, consider tests formed from a trace t of SP, and a forbidden continuation e, that is, an event e that is not in the set of initials of SP after the trace t. On the other hand, as explained later on in Section 5, to construct the exhaustive test set for *conf*, which is concerned with the reduction of deadlocks, we need the sets of acceptances.

Here, we give symbolic characterisations of sets of traces, initials (after a given trace), and acceptances (also after a given trace) of processes and actions. We also discuss the instantiation operations used to relate the symbolic and the concrete sets characterised by the denotational semantics.

3.1 Traces

To represent a group of traces that record communications over the same channels in the same order, we define constrained symbolic traces.

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 $\mathtt{st}$: the alphabet $\alpha\mathtt{st}$ of $\mathtt{st}$ defined inductively as follows.

$$\alpha\langle\,\rangle = \langle\,\rangle \qquad \alpha(\langle \mathtt{d}.\alpha_0\rangle \frown \mathtt{st}) = \langle \alpha_0 \rangle \frown \alpha\mathtt{st}$$

We emphasise that the variables in the alphabet of a trace are symbols. They are not variables used in the state of any *Circus* model, but names used to represent communicated values along the interactions.

Our treatment of alphabet is akin to that in the UTP. The only difference is that, to simplify definitions, we take it to be a sequence, rather than a set. It is an unbounded sequence of fresh names, with no repetitions, that records symbolic variables used to represent communicated values. The symbolic traces $\mathtt{st}$ over an alphabet $\mathtt{a}$ are those for which $\alpha\mathtt{st}$, as defined above, is a prefix of $\mathtt{a}$, that is, $\alpha\mathtt{st} \leq \mathtt{a}$, where $\leq$ is the prefix relation.

Our sets of constrained symbolic traces, as defined below (as well as the sets of constrained symbolic initial and acceptances, defined later) are parametrised by an alphabet $\mathtt{a}$. It fixes the variables that can be used in a symbolic trace, and the order in which they can be used. (We observe that the order of the variables in the alphabet is similar to version numbers of variable valuations during a symbolic program execution.)

For a process $\mathtt{begin\ state\ [x : T] \bullet A\ end}$, the constrained symbolic traces over an alphabet $\mathtt{a}$ are those of its main action $\mathtt{A}$, starting from a state in which 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 below using the operational semantics. The traces are finite, and use only a portion of the variables in $\mathtt{a}$.

Definition 1

$$\begin{array}{l} cstraces^{\mathtt{a}}(\mathtt{begin\ state[x : T] \bullet A\ end}) = \\ \quad cstraces^{\mathtt{a}}(\mathtt{w_0} \in \mathtt{T}, \mathtt{x} := \mathtt{w_0}, \mathtt{A}) \end{array}$$

$$\begin{array}{l} cstraces^{\mathtt{a}}(\mathtt{c_1}, \mathtt{s_1}, \mathtt{A_1}) = \\ \quad \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c_2}, \mathtt{s_2}, \mathtt{A_2} \mid \alpha\mathtt{st} \leq \mathtt{a} \wedge (\mathtt{c_1} \mid \mathtt{s_1} \models \mathtt{A_1}) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c_2} \mid \mathtt{s_2} \models \mathtt{A_2}) \\ \bullet\ (\mathtt{st}, \exists (\alpha\mathtt{c_2} \setminus \alpha\mathtt{st}) \bullet \mathtt{c_2}) \end{array} \right\} \end{array}$$

□

The parameter $\mathtt{a}$ determines the alphabet of the (constrained) symbolic traces. A constrained symbolic trace $(\mathtt{st}, \mathtt{c})$ is said to be over an alphabet $\mathtt{a}$ if its symbolic trace $\mathtt{st}$ is over $\mathtt{a}$, as defined below.

Symbolic variables used in the evaluation of the operational semantics to represent internal values of the state are not included in the alphabet. As said above, $\mathtt{a}$ contains variables that denote values that are visible in the observation of a process. For clarity, in examples, we use α_0, α_1, and so on,

$$\frac{}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}) \xRightarrow{\langle\,\rangle} (\mathtt{c} \mid \mathtt{s} \models \mathtt{A})} \qquad \frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xRightarrow{\langle\,\rangle} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}$$

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d}?\alpha_0} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xRightarrow{\langle\,\mathtt{d}.\alpha_0\,\rangle} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)} \qquad \frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d}!\alpha_0} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xRightarrow{\langle\,\mathtt{d}.\alpha_0\,\rangle} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)}$$

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

Table 1 Annotated transition relation

for variables in the alphabet, and w_0, w_1, and so on, for variables that can be either in the alphabet or represent an internal value of the state.

The transition relation annotated with a trace is defined in terms of the transition relation that defines the operational semantics in the usual way. All the rules for this new relation are presented in Table 1. It ignores the decorations in the labels that record whether they represent inputs or outputs, since this information is not relevant for the definition of traces.

The constraint $\exists (\alpha\mathtt{c}_2 \setminus \alpha\mathtt{st}) \bullet \mathtt{c}_2$ is obtained from that built using the operational semantics, namely $\mathtt{c}_2$, by quantifying its free (symbolic) variables that are not in the alphabet $\alpha\mathtt{st}$ of the associated trace $\mathtt{st}$. These are the symbolic variables used in the operational semantics to represents internal values of state or local variables (as opposed to those in $\alpha\mathtt{st}$, which represent communications). We use $\alpha\mathtt{c}$ to denote the set of free variables of a constraint $\mathtt{c}$. We adopt the standard notion of free variables for (texts of) predicates: $\mathtt{c}$ is (the text of) a predicate on symbolic variables, as explained previously.

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. Instantiation of a constrained symbolic trace is defined later on in this section.

By fixing the alphabet of interest, we avoid complications when dealing with, or comparing, symbolic entities built using different symbols to represent the same values. Typically, this would require renaming as in [25].

In contrast, the order of the variables of the alphabet determines a unique way in which equivalent symbolic traces are expressed. If we use a particular alphabet, to conclude, for example, that two constrained symbolic traces $(\mathtt{st}_1, \mathtt{c}_1)$ and $(\mathtt{st}_2, \mathtt{c}_2)$ are equivalent, we need to check only that $\mathtt{st}_1 = \mathtt{st}_2$ and $c_1 \Leftrightarrow c_2$; no renaming is needed. Fixing the alphabet in $cstraces^{\mathsf{a}}(\mathtt{P})$, for example, fixes the representation of symbolic traces in this set.

Example 1 NoOverlap and *Overlap* These are simple processes, with no state, but with local variables introduced by input prefixings. They are inspired by examples originally presented in [22, Example 4.2].

$$\begin{array}{l}\textbf{process}\ \mathit{NoOverlap} \mathrel{\widehat=} \textbf{begin}\ \bullet\\ \quad g?x : x > 10 \to g!x \to \textbf{Stop} \mathrel{\Box} g?x : x < 10 \to h!x \to \textbf{Stop}\\ \textbf{end}\end{array}$$

This example is interesting because, initially, the process *NoOverlap* does not accept a communication of the value 10 in the channel g, and also depending on whether the value communicated is greater or less than 10, it can be followed by another communication of the same value on g or on another channel h. We present below the set of constrained symbolic traces of *NoOverlap* for an alphabet $\mathtt{a}$ such that $\langle \alpha_0, \alpha_1 \rangle \leq \mathtt{a}$. In this example, and in others to follow, for the sake of conciseness and clarity, we omit the information about the types of the channels. Strictly speaking, it needs to be recorded in the constraints on the values communicated through them.

$$\begin{array}{l} \mathit{cstraces}^{\mathtt{a}}(\mathtt{NoOverlap}) =\\ \quad \{\ (\langle\,\rangle, \mathtt{true}),\\ \quad\quad (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10}), (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10}),\\ \quad\quad (\langle \mathtt{g}.\alpha_0, \mathtt{g}.\alpha_1 \rangle, \alpha_0 > \mathtt{10} \wedge \alpha_1 = \alpha_0), (\langle \mathtt{g}.\alpha_0, \mathtt{h}.\alpha_1 \rangle, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0)\ \} \end{array}$$

We observe that, in examples, we do not keep the textual representation of constraints exactly as generated by the operational semantics and identified in Definition 1 and in others to follow. Instead, for clarity, we give the text of an equivalent predicate that captures the constraints in a more concise way.

A possible optimisation is the representation of $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})$ and $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$ by a single trace $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10} \vee \alpha_0 < \mathtt{10})$. The introduction of this sort of optimisation, however, complicates definitions and proofs. We observe, for example, that the different traces above have different continuations, and this is naturally identified directly from the operational semantics in Definition 1 (without any need for normalisation of predicates, for instance). If the optimised representation proves to be of practical interest, it is not difficult to justify it separately.

For the process *Overlap* below, the restrictions on the values communicated on g are not mutually exclusive. For values between 5 and 10, both a second communication on g or a communication on h are possible.

$$\begin{array}{l}\textbf{process}\ \mathit{Overlap} \mathrel{\widehat=} \textbf{begin}\ \bullet\\ \quad g?x : x < 10 \to g!x \to \textbf{Stop} \mathrel{\Box} g?x : x > 5 \to h!x \to \textbf{Stop}\\ \textbf{end}\end{array}$$

The set of constrained symbolic traces of *Overlap* is similar.

$$\begin{array}{l} \mathit{cstraces}^{\mathtt{a}}(\mathtt{Overlap}) =\\ \quad \{\ (\langle\,\rangle, \mathtt{true}),\\ \quad\quad (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10}), (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{5}),\\ \quad\quad (\langle \mathtt{g}.\alpha_0, \mathtt{g}.\alpha_1 \rangle, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0), (\langle \mathtt{g}.\alpha_0, \mathtt{h}.\alpha_1 \rangle, \alpha_0 > \mathtt{5} \wedge \alpha_1 = \alpha_0)\ \} \end{array}$$

provided $\langle \alpha_0, \alpha_1 \rangle \leq \mathtt{a}$. $\Box$

The set of traces of a process can be obtained from its set of constrained symbolic traces by instantiation, as defined by the function instT below.

Definition 2 $\text{instT}(\mathtt{st}, \mathtt{c}) = \{\, v \mid c[v/\alpha st] \bullet st[v/\alpha st] \,\}$ □

We use $c[v/\alpha st]$ to denote the substitution in c of all occurrences of the variables in αst with the corresponding value in the list v. Similarly, in $st[v/\alpha st]$ we apply substitution to the symbolic trace st and get a concrete trace. In this case, substitution is for the symbolic variables in st: those in αst, which we define to be free in the symbolic trace st.

In the sequel, we also write substitutions like $c[v_0, v/\alpha_0, \alpha st]$, where the list of values is enriched with the value v_0 at the beginning, and the list of variables with α_0. Therefore, α_0 is replaced with v_0, and all the other variables in αst with the corresponding values in v as before. For $\text{instT}(\mathtt{st}, \mathtt{c})$ to be well defined, all free variables of $\mathtt{c}$ have to be in $\alpha\mathtt{st}$.

Instantiation of a trace can be done one event at a time, but each choice of a value for a symbolic variable has to be recorded in the constraint. This is established below in Lemma 1; its proof can be found in [15], as can all other omitted proofs in this and in the next section.

Lemma 1 Partial instantiation - traces

$$\text{instT}(\langle \mathtt{d}.\alpha_0 \rangle \frown \mathtt{st}, \mathtt{c}) = \{\, v_0, t \mid t \in \text{instT}(\mathtt{st}, \mathtt{c}[\mathtt{v}_0/\alpha_0]) \bullet \langle d.v_0 \rangle \frown t \,\}$$

provided $\alpha_0 \notin \alpha\mathtt{st}$. □

The following proposition formalises the relationship between symbolic and concrete traces induced by the instantiation function instT.

Proposition 1 *For every alphabet* $\mathtt{a}$*,*

$$\mathit{traces}(\mathtt{P}) = \bigcup \{\, \mathtt{cst} : \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}) \bullet \text{instT}(\mathtt{cst}) \,\}$$

□

By properties of the operational semantics, for all $(\mathtt{st}, \mathtt{c})$ in $\mathit{cstraces}^{\mathtt{a}}(\mathtt{P})$, the free variables of $\mathtt{c}$ are in $\alpha\mathtt{st}$, so that $\text{instT}(\mathtt{st}, \mathtt{c})$ is well defined.

The results in this section are used in Sections 4 and 5.

3.2 Initials

To define symbolic tests, we also need to provide a symbolic definition of the initials of a process. The set $\mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ contains the constrained symbolic events that represent valid continuations of the constrained symbolic trace $(\mathtt{st}, \mathtt{c})$ over the alphabet $\mathtt{a}$. It is defined just for the constrained symbolic traces $(\mathtt{st}, \mathtt{c})$ of $\mathtt{P}$ over $\mathtt{a}$. The constrained symbolic events are pairs formed by a symbolic event, and a constraint that refers to the symbolic variable of the event, as well as those in the alphabet of $\mathtt{st}$.

Definition 3

$$\mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c})) =$$
$$\{\mathtt{se}, \mathtt{c}_1 \mid (\mathtt{st} \frown \langle \mathtt{se} \rangle, \mathtt{c}_1) \in \mathit{cstraces}^{\mathtt{a}(\mathtt{P})} \wedge (\exists\, a \bullet c \wedge c_1) \bullet (\mathtt{se}, \mathtt{c} \wedge \mathtt{c}_1)\,\}$$

provided $(\mathtt{st}, \mathtt{c}) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{P})$. □

We require that the symbolic events $\mathtt{se}$ in $\mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ are continuations of $\mathtt{st}$ compatible with $\mathtt{c}$, that is, such that $(\exists\, a \bullet c \wedge c_1)$. We use the alphabet $\mathtt{a}$ to construct the existential quantification that states that there is a valuation of the symbolic variables (of the alphabet $\mathtt{a}$) that satisfies both c and c_1. In addition, in the constrained symbolic event, $\mathtt{c}$ is recorded to characterise the fact that the symbolic event is a continuation of a trace constrained by $\mathtt{c}$. We note that both c and c_1 are constraints on the variables of the alphabet $\mathtt{a}$, which is a parameter in $csinitials^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$.

Example 2 NoOverlap and *Overlap* First of all, we consider $\mathtt{NoOverlap}$ and the trace $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})$. We omit the alphabet for conciseness.

$$csinitials(\mathtt{NoOverlap}, (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})) = \{(\mathtt{g}.\alpha_1, \alpha_0 > \mathtt{10} \wedge \alpha_1 = \alpha_0)\}$$

The trace $(\langle \mathtt{g}.\alpha_0, \mathtt{h}.\alpha_1 \rangle, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0)$ does not give rise to another constrained symbolic event in $csinitials(\mathtt{NoOverlap}, (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10}))$, even though the trace $\langle \mathtt{g}.\alpha_0, \mathtt{h}.\alpha_1 \rangle$ of course extends $\langle \mathtt{g}.\alpha_0 \rangle$. This is because the conjunction $\alpha_0 > 10 \wedge \alpha_0 < 10 \wedge \alpha_1 = \alpha_0$ is false.

For $\mathtt{Overlap}$ and the trace $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$ we have the following.

$$\begin{array}{l} csinitials(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})) = \\ \quad \{(\mathtt{g}.\alpha_1, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0), (\mathtt{h}.\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0)\} \end{array}$$

In this case, the constrained symbolic trace $(\langle \mathtt{g}.\alpha_0, \mathtt{h}.\alpha_1 \rangle, \alpha_0 > \mathtt{5} \wedge \alpha_1 = \alpha_0)$ gives rise to a symbolic event $\mathtt{h}.\alpha_1$, but the constraint of this trace is strengthened with $\alpha_0 < \mathtt{10}$ to record that only some of the traces represented by the constrained symbolic trace are of interest. □

As for symbolic and concrete traces, instantiation can be used to determine the concrete initials of a process in terms of its set of constrained symbolic initials. Instantiation of a symbolic initial, however, needs to take into account the trace that leads to it since the constraint restricts the symbolic variables used to represent all the previously communicated values.

We introduce, first, the notion of a trace property in relation to an alphabet. A concrete trace defines actual events, and, therefore, communicated values; in this way, it can be seen as a definition of the values of the symbolic variables of an associated alphabet. The syntactic function p defined below determines the predicate or, more precisely, the conjunction of equalities, defined by a trace t over the variables of a given alphabet $\mathtt{a}$.

$$\mathtt{p}^{\mathsf{a}}(\langle\rangle) = \mathtt{true} \qquad \mathtt{p}^{(\langle \alpha_0 \rangle ^\frown \mathsf{a})}(\langle d.v \rangle ^\frown t) = (\alpha_0 = \mathtt{v} \wedge \mathtt{p}^{\mathsf{a}}(t))$$

As already said, in a symbolic event $(\mathtt{d}.\alpha_0, \mathtt{c})$, the constraint $\mathtt{c}$ restricts the value of α_0, and perhaps those of other variables of an alphabet $\mathtt{a}$ to keep track of the relationship between α_0 and other values communicated previously. If the trace t that leads to the event is known, we can strengthen $\mathtt{c}$ to take t into account. This is the purpose of the new operator below.

$$(\mathtt{d}.\alpha_0, \mathtt{c}) \restriction^{\mathsf{a}} t = (\mathtt{d}.\alpha_0, \mathtt{c} \wedge \mathtt{p}^{\mathsf{a}}(t))$$

We often use relational image to apply this operator to a set of initials.

Below, we characterise instantiation of a symbolic event $(\mathtt{d}.\alpha_0, \mathtt{c})$ using a function from symbolic events to sets of concrete events.

Definition 4 $\mathrm{instE}^{\mathtt{a}}(\mathtt{d}.\alpha_0, \mathtt{c}) = \{v_0, v \mid c[v_0, v/\alpha_0, a] \bullet d.v_0\}$ □

Instantiation takes into account the restrictions of c over all variables of the alphabet a as well as the variable α_0 used in the event. Here, and in the sequel, we use v to stand for a (list of) value(s); v_0 stands for a single value.

The lemma below relates instantiation of constrained symbolic traces, as defined in the previous section, and events. It states that, if we can instantiate a symbolic trace $\mathtt{st} \frown \langle \mathtt{d}.\alpha_0 \rangle$ to obtain a concrete trace $t \frown \langle e \rangle$, then we can get the same event e corresponding to $\mathtt{d}.\alpha_0$ if we first constrain the symbolic event to the concrete trace t and then instantiate the result.

Lemma 2 Trace and event instantiation

$$t \frown \langle e \rangle \in \mathrm{instT}(\mathtt{st} \frown \langle \mathtt{d}.\alpha_0 \rangle, \mathtt{c}) \Leftrightarrow (e \in \mathrm{instE}^{\alpha \mathtt{st}}((\mathtt{d}.\alpha_0, \mathtt{c}) \upharpoonright^{\alpha \mathtt{st}} t))$$

□

The following proposition relates symbolic and concrete sets of initials. It describes how, for a process P, and a trace t, we can obtain the set *initials*(P, t) of concrete initials of P after t using the symbolic sets of initials.

Proposition 2 *For every alphabet a,*

$$\begin{aligned} \mathit{initials}(\mathtt{P}, t) = \bigcup \{ & \mathtt{st}, \mathtt{c} \mid (\mathtt{st}, \mathtt{c}) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}) \land t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}) \\ & \bullet \bigcup \mathrm{instE} \, (\!| \, (\!| \mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c})) \, |\!) \upharpoonright^{\mathtt{a}} t \, |\!) \\ \} & \end{aligned}$$

□

For each of the symbolic traces (st, c) of P that can be instantiated to t, we strengthen the constraint of the events in $\mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ to take t into account, and instantiate them. If t is not a trace of P, then it is not an instance of any of its constrained symbolic traces, and *initials*(P, t) is empty.

In fact, it is enough to consider one constrained symbolic trace (st, c) that can be instantiated to t. First, we observe that a symbolic trace st may occur in more than one constrained symbolic trace of $\mathit{cstraces}^{\mathtt{a}}(\mathtt{P})$. In the case of *NoOverlap* and *Overlap*, for instance, $\langle \mathtt{g}.\alpha_0 \rangle$ is in two traces. In addition, the constraints in such traces may not be mutually exclusive, as in the *Overlap* example. Therefore, it is possible that a trace t may be an instance of more than one constrained symbolic trace in $\mathit{cstraces}^{\mathtt{a}}(\mathtt{P})$. If, however, t is indeed an instance of two traces $(\mathtt{st}, \mathtt{c}_1)$ and $(\mathtt{st}, \mathtt{c}_2)$, the initials for $(\mathtt{st}, \mathtt{c}_1)$ and $(\mathtt{st}, \mathtt{c}_2)$, when restricted to t, are the same. This is formalised below.

Proposition 3 *For every process* P *and alphabet* a*,*

$$\begin{aligned} & \left(\begin{array}{l} (\mathtt{st}, \mathtt{c}_1) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}) \land (\mathtt{st}, \mathtt{c}_2) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{P}) \land \\ t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_1) \land t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_2) \end{array} \right) \Rightarrow \\ & \quad (\!| \mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}_1)) \, |\!) \upharpoonright^{\mathtt{a}} t = (\!| \mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}_2)) \, |\!) \upharpoonright^{\mathtt{a}} t \end{aligned}$$

□

More specifically, this states that continuations of t calculated from any of the constrained symbolic traces that represent t are consistent.

Example 3 Overlap The trace $\langle g.6\rangle$ of *Overlap* is an instance of two of its constrained symbolic traces: $(\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10})$ and $(\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 > \mathtt{5})$. Therefore, we can use either of the sets $\mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10}))$ or $\mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 > \mathtt{5}))$ to determine $\mathit{initials}(\mathtt{Overlap}, \langle \mathtt{g.6}\rangle)$. As said before, the first set of constrained symbolic initials is as follows.

$$\begin{array}{l} \mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10})) = \\ \quad \{(\mathtt{g}.\alpha_1, \alpha_0 < \mathtt{10} \land \alpha_1 = \alpha_0), (\mathtt{h}.\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \land \alpha_1 = \alpha_0)\} \end{array}$$

The restriction of its events to $\langle g.6\rangle$ determines the value of α_0 to be 6.

$$\begin{array}{l} (\!|\, \mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10})) \,|\!) \restriction \langle g.6\rangle = \\ \quad \{(\mathtt{g}.\alpha_1, \alpha_0 = \mathtt{6} \land \alpha_1 = \mathtt{6}), (\mathtt{h}.\alpha_1, \alpha_0 = \mathtt{6} \land \alpha_1 = \mathtt{6})\} \end{array}$$

On the other hand, $\mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 > \mathtt{5}))$ is as follows.

$$\begin{array}{l} \mathit{csinitials}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 > \mathtt{5})) = \\ \quad \{(\mathtt{g}.\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \land \alpha_1 = \alpha_0), (\mathtt{h}.\alpha_1, \mathtt{5} < \alpha_0 \land \alpha_1 = \alpha_0)\} \end{array}$$

The restriction of this set to $\langle g.6\rangle$ is the same as above. Instantiation of the events of these sets gives $\{g.6, h.6\}$ as expected. □

As said at the beginning of this section, we use the complement of the set of initials in the construction of tests for traces refinement. Because of the symbolic nature of $\mathit{csinitials}^{\mathsf{a}}(\mathtt{P}, \mathtt{cst})$, however, it is not meaningful to consider its complement: it does not characterise the complement of the concrete set of initials. Instead, we define a different set $\overline{\mathit{csinitials}}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ that contains the 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 $(\mathtt{st}, \mathtt{c})$. A constrained symbolic event $(\mathtt{se}, \mathtt{c})$ is over an alphabet $\mathtt{a}$ if the free variables of $\mathtt{c}$ are in a prefix of $\mathtt{a}$ that finishes with the symbolic variable used in $\mathtt{se}$.

Definition 5

$$\begin{array}{l} \overline{\mathit{csinitials}}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c})) = \\ \quad \left\{ \begin{array}{l} \mathtt{d}, \alpha_0, \mathtt{c}_1 \mid \\ \quad \left(\begin{array}{l} \alpha_0 = \mathtt{a}(\#\,\mathtt{st} + \mathtt{1}) \land \\ \mathtt{c}_1 = \mathtt{c} \land \neg \bigvee \{\mathtt{c}_2 \mid (\mathtt{d}.\alpha_0, \mathtt{c}_2) \in \mathit{csinitials}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))\,\} \end{array} \right) \\ \bullet\ (\mathtt{d}.\alpha_0, \mathtt{c}_1) \end{array} \right\} \end{array}$$

provided $(st, c) \in \mathit{cstraces}^a(P)$. □

To define the symbolic events, we use the next variable α_0 in the alphabet $\mathtt{a}$. If $\mathit{csinitials}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ is empty, then the disjunction used to define the constraints $\mathtt{c}_1$ of the symbolic events is $\mathtt{false}$, and therefore $c_1 = c$. On the other hand, if $\mathit{csinitials}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ includes several symbolic events over a channel d, then a single event is included in $\overline{\mathit{csinitials}}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ to represent communications over d; its constraint guarantees that $\mathtt{c}$ holds, but none of the constraints of the symbolic initials in $\mathit{csinitials}^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ does.

Example 4 NoOverlap We consider the constrained symbolic trace $(\langle\,\rangle, \mathtt{true})$ of *NoOverlap*, and observe that $\mathit{csinitials}^{\mathsf{a}}(\mathtt{NoOverlap}, (\langle\,\rangle, \mathtt{true}))$ is the set $\{\,(\mathtt{g}.\alpha_0, \alpha_0 > \mathtt{10}), (\mathtt{g}.\alpha_0, \alpha_0 < \mathtt{10})\,\}$. Therefore, no communication over h is in the set of initials nor is a communication over g of the value 10.

$$\overline{csinitials}^{\mathtt{a}}(\mathtt{NoOverlap}, (\langle\,\rangle, \mathtt{true})) = \{\, (\mathtt{h}.\alpha_0, \mathtt{true}), (\mathtt{g}.\alpha_0, \alpha_0 = \mathtt{10}) \,\}$$

Similarly, for the trace $(\langle\, \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})$, we have the following result.

$$\begin{aligned}&\overline{csinitials}^{\mathtt{a}}(\mathtt{NoOverlap}, (\langle\, \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})) = \\ &\quad \{\, (\mathtt{g}.\alpha_1, \alpha_0 > \mathtt{10} \wedge \alpha_1 \neq \alpha_0), (\mathtt{h}.\alpha_1, \alpha_0 > \mathtt{10}) \,\}\end{aligned}$$

In this case, $(\mathtt{g}.\alpha_1, \alpha_0 > \mathtt{10} \wedge \alpha_1 = \alpha_0)$ and $(\mathtt{h}.\alpha_1, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0)$ are the symbolic initials. If we negate their constraints, and conjoin with $\alpha_0 > \mathtt{10}$, we get the set already shown above. $\Box$

Using Proposition 2, we can prove that this set does indeed define the complement of the set of initials. This is established by the lemma below.

Lemma 4 Instantiation of $\overline{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ *For every alphabet* $\mathtt{a}$*,*

$$\begin{aligned}\overline{initials(\mathtt{P}, t)} = \bigcup \{\, &\mathtt{st}, \mathtt{c} \mid (\mathtt{st}, \mathtt{c}) \in cstraces^{\mathtt{a}}(\mathtt{P}) \wedge t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}) \\ &\bullet \bigcup \mathrm{instE} \;(\!|\; (\!|\, \overline{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c})) \,|\!) \restriction^{\mathtt{a}} t \,|\!) \\ \}\end{aligned}$$

provided $t \in traces(\mathtt{P})$*.* $\Box$

This result allows us to overcome the difficulty of handling negation when working with symbolic characterisations of sets.

3.3 Acceptances

As already explained, to construct tests for *conf*, we need sets of acceptances. The characterisation of failures in terms of the CSP operational semantics presented in [58, p.189], and the standard characterisation of acceptances in terms of failures are the inspiration for our definition of constrained symbolic acceptances. The calculation below gives a clear indication of how acceptances of a process P after one of its traces t can be defined in terms of configurations of the CSP operational semantics and sets of initials. (The treatment of termination in *Circus* is different from that in CSP, which uses a special event ✓. The definition of failures used below is, therefore, a simplification of that in [58], since there are no *Circus* traces with a ✓ event.)

$acceptances(P, t)$

$= \{\, X \mid (t, X) \notin failures(P) \,\}$

[definition of *acceptances* (in terms of *failures*)]

$= \{\, X \mid \neg\, (\exists\, Q \mid P \overset{t}{\Longrightarrow} Q \wedge Q \textit{ is stable} \bullet initials(Q) \cap X = \varnothing) \,\}$

[definitions of *failures*(P) and *initials*(Q) as presented in [58]]

$= \{\, X \mid (\forall\, Q \mid P \overset{t}{\Longrightarrow} Q \wedge Q \textit{ is stable} \bullet initials(Q) \cap X \neq \varnothing) \,\}$

[predicate calculus]

$$= \{ X \mid (\forall Q \mid P \overset{t}{\Longrightarrow} Q \land Q \textit{ is stable} \bullet (\exists e : X \bullet e \in \textit{initials}(Q))) \}$$

[property of sets]

To summarise, in CSP, we have the result below, which is the basis for our characterisation of symbolic acceptances as explained in the following.

$$\begin{array}{l} \textit{acceptances}(P, t) = \\ \quad \{ X \mid (\forall Q \mid P \overset{t}{\Longrightarrow} Q \land Q \textit{ is stable} \bullet (\exists e : X \bullet e \in \textit{initials}(Q))) \} \end{array} \tag{7}$$

In the *Circus* operational semantics, as explained in Section 2.2.2, the events e are represented by labels $\mathtt{d}?\alpha_0$ and $\mathtt{d}!\alpha_0$. To capture acceptances from sets of symbolic events, we rely on this distinction between inputs and outputs.

The set $\textit{csacceptances}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ of constrained symbolic acceptances over the alphabet $\mathtt{a}$ of a process $\mathtt{P}$ after a constrained symbolic trace $(\mathtt{st}, \mathtt{c})$ is a set of constrained symbolic events (over $\mathtt{a}$) with associated input and output information. In order to define the set $\textit{csacceptances}^{\mathtt{a}}(\mathtt{P}, \mathtt{cst})$, we use the set $\textit{IOcsinitials}^{\mathtt{a}}_{\mathtt{st}}(\mathtt{c}, \mathtt{s}, \mathtt{A})$ of constrained symbolic initials over $\mathtt{a}$ for an action $\mathtt{A}$ (in the context of a constraint $\mathtt{c}$ and a state assignment $\mathtt{s}$, and after a trace $\mathtt{st}$) that record input and output information.

The set $\textit{IOcsinitials}^{\mathtt{a}}_{\mathtt{st}}(\mathtt{c}, \mathtt{s}, \mathtt{A})$ is defined below

Definition 6

$$\begin{array}{l} \textit{IOcsinitials}^{\mathtt{a}}_{\mathtt{st}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1) = \\ \left\{ \begin{array}{l} \{\mathtt{l}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid \\ \quad (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \land \operatorname{chan} \mathtt{l} \neq \epsilon \land \alpha(\mathtt{st} \frown \langle \mathtt{l} \rangle) \leq \mathtt{a} \\ \bullet (\mathtt{l}, \exists (\alpha \mathtt{c}_2 \setminus (\alpha(\mathtt{st} \frown \langle \mathtt{l} \rangle)) \bullet \mathtt{c}_2) \end{array} \right\} \end{array}$$

□

The initials in $\textit{IOcsinitials}^{\mathtt{a}}_{\mathtt{st}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1)$ are called here input-output constrained symbolic initials (over $\mathtt{a}$). They are determined by the labels $\mathtt{l}$ of non-silent ($\operatorname{chan} \mathtt{l} \neq \epsilon$) transitions from $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$ whose symbolic variable is determined by the alphabet $\mathtt{a}$. It is the next variable in $\mathtt{a}$ after those used in $\mathtt{st}$; this is guaranteed by $\alpha(\mathtt{st} \frown \langle \mathtt{l} \rangle) \leq \mathtt{a}$. The constraint $\mathtt{c}_2$ of the target configuration of each transition determines the constraint of the corresponding symbolic event. The variables not in the alphabet of $\mathtt{st} \frown \langle \mathtt{l} \rangle$, that is, those that do not represent communicated values, are quantified. Strictly speaking, $\mathtt{st} \frown \langle \mathtt{l} \rangle$ is not a symbolic trace as defined in Section 3.1, since $\mathtt{l}$ includes an input or output decoration. We note, however, that the extension of the definition of $\alpha \mathtt{st}$ to cater for such decorations is trivial.

Previously, we defined the set $\textit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c}))$ of initials of a process $\mathtt{P}$ after a particular trace $(\mathtt{st}, \mathtt{c})$. Here, to characterise acceptances, we need to define the set of initials of a particular action: we define $\textit{IOcsinitials}^{\mathtt{a}}_{\mathtt{st}}(\mathtt{c}, \mathtt{s}, \mathtt{A})$ for a particular action $\mathtt{A}$, and identify its context explicitly via the constraint $\mathtt{c}$ and state $\mathtt{s}$. Comparatively, in (7), the particular processes Q of interest arise from the execution of P along a trace t. In general, it is not possible to uniquely identify Q using P and t, since, due to nondeterminism, t may lead to several different states of P. We need to consider them all, and identify the initials of each of them. This is reflected below in Definition 7.

Example 5 Overlap After an event $g.\alpha_0$ of the trace $(\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10})$, if $\alpha_0 > 5$, then *Overlap* may be ready to behave like either $g!x \to \mathbf{Stop}$ or $h!x \to \mathbf{Stop}$. The trace itself does not identify a single action, but the fact that either of them is possible. To characterise the acceptance set of *Overlap* after $(\langle \mathtt{g}.\alpha_0\rangle, \alpha_0 < \mathtt{10})$, we need to know the set of initials of $g!x \to \mathbf{Stop}$ and of $h!x \to \mathbf{Stop}$. As indicated by the calculation above, we need at least one element of each of them in an acceptance set. □

As for traces, the acceptances of a process after a trace are defined in terms of those for its main action. In both cases, the acceptances are well defined only for traces of the process or action.

Definition 7

$$
\begin{array}{l}
\mathit{csacceptances}^{\mathsf{a}}(\mathtt{begin\ state}\,[\,\mathtt{x}:\mathtt{T}\,] \bullet \mathtt{A\ end}, (\mathtt{st},\mathtt{c})) = \\
\quad \mathit{csacceptances}^{\mathsf{a}}(\mathtt{w}_0 \in \mathtt{T}, \mathtt{x} := \mathtt{w}_0, \mathtt{A}, (\mathtt{st},\mathtt{c}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{csacceptances}^{\mathsf{a}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1, (\mathtt{st},\mathtt{c})) = \\
\left\{ SX \mid \left(\begin{array}{l}
\forall\, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid \\
\qquad \left(\begin{array}{l} (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \overset{\mathtt{st}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \wedge \\ (\exists\, a \bullet c_2 \wedge c) \wedge \mathit{stable}(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \end{array} \right) \bullet \\
\qquad \exists\, \mathtt{iose} : SX \bullet \mathtt{iose} \in \mathit{IOcsinitials}^{\mathsf{a}}_{\mathtt{st}}(\mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2) \restriction \mathtt{c}
\end{array} \right) \right\}
\end{array}
$$

provided we have

$$
\begin{array}{l}
\mathtt{cst} \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{begin\ state}\,[\,\mathtt{x}:\mathtt{T}\,] \bullet \mathtt{A\ end}) \\
(\mathtt{st},\mathtt{c}) \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1)
\end{array}
$$

and where

$$
\mathit{stable}(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) = \neg\, \exists\, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \bullet (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)
$$

□

As indicated by (7), to define the sets SX of symbolic acceptances of an action $\mathtt{A}_1$ (in the context of a constraint $\mathtt{c}_1$ and a state $\mathtt{s}_1$) after a trace $(\mathtt{st},\mathtt{c})$, we consider all stable configurations $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ that can be reached from $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$. For each of them, we require SX to include at least one element of its set $\mathit{IOcsinitials}^{\mathsf{a}}_{\mathtt{st}}(\mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2)$ of initials.

Since the trace $(\mathtt{st},\mathtt{c})$ is symbolic, we need to make sure that the configuration $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ reached using $\mathtt{st}$ is compatible with $\mathtt{c}$, that is, $(\exists\, a \bullet c_2 \wedge c)$. As before, we use the alphabet $\mathtt{a}$ to construct the existential quantification that requires that there is a valuation satisfying both c_2 and c. In addition, the events in the set $\mathit{IOcsinitials}^{\mathsf{a}}_{\mathtt{st}}(\mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2)$ need to be further constrained by $\mathtt{c}$. For that, we use above the operator $SX \restriction \mathtt{c}$, which we define below. It yields the set of input-output constrained symbolic events that can be obtained by strengthening the constraint of those in SX with $\mathtt{c}$. Events $(\mathtt{iose}, \mathtt{c}_1)$ that are not compatible with $\mathtt{c}$ are eliminated; for these, the property defined by $(\exists\, a \bullet c \wedge c_1)$ does not hold.

$$
SX \restriction \mathtt{c} = \{\mathtt{iose}, \mathtt{c}_1 \mid (\mathtt{iose}, \mathtt{c}_1) \in SX \wedge (\exists\, a \bullet c \wedge c_1) \bullet (\mathtt{iose}, \mathtt{c} \wedge \mathtt{c}_1)\}
$$

This operator is used above because the set $\mathit{IOcsinitials}^{\mathsf{a}}_{\mathtt{st}}(\mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2)$ of constrained symbolic initials from $(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ does not record that we are interested only in the traces that satisfy the constraint c.

A stable configuration is one from which there are no silent transitions available. In our operational semantics, these have the label ϵ, as formalised in the definition of $\mathit{stable}(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$ above.

Example 6 Overlap We consider the set of constrained symbolic acceptances of *Overlap* after the trace $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$. There are two configurations that can be reached with $\langle \mathtt{g}.\alpha_0 \rangle$. In one of them the constraint is $\alpha_0 < \mathtt{10}$ and in the other $\alpha_0 > \mathtt{5}$. Both of these constraints are compatible with that in the symbolic trace, that is, $\alpha_0 < \mathtt{10}$, and both configurations are stable. Therefore, we need to consider the sets of input-output constrained symbolic initials for both configurations. They are presented below. In examples, we omit both the `a` and `st` parameters of *IOcsinitials*.

$$\mathit{IOcsinitials}(\alpha_0 < \mathtt{10}, \mathtt{x} := \alpha_0, \mathtt{g!x} \rightarrow \mathtt{Stop}) = \{ (\mathtt{g!}\alpha_1, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0) \} \tag{8}$$

$$\mathit{IOcsinitials}(\alpha_0 > \mathtt{5}, \mathtt{x} := \alpha_0, \mathtt{h!x} \rightarrow \mathtt{Stop}) = \{ (\mathtt{h!}\alpha_1, \mathtt{5} < \alpha_0 \wedge \alpha_1 = \alpha_0) \} \tag{9}$$

The event in (9) needs to be constrained to reflect the fact that the trace of interest has the constraint $\alpha_0 < \mathtt{10}$. Filtering gives the result below.

$$\begin{array}{l} \mathit{IOcsinitials}(\alpha_0 > \mathtt{5}, \mathtt{x} := \alpha_0, \mathtt{h!x} \rightarrow \mathtt{Stop}) \upharpoonright \alpha_0 < \mathtt{10} = \\ \quad \{ (\mathtt{h!}\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0) \} \end{array} \tag{10}$$

Filtering with $\alpha_0 < \mathtt{10}$ does not change the set (8). In conclusion, the set $\mathit{csacceptances}(\mathtt{Overlap}, (\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10}))$ contains all the sets of constrained symbolic events that include (8) and (10). □

To obtain sets of concrete acceptances using the sets of constrained symbolic acceptances we need to distinguish input and output symbolic events.

– A symbolic variable in an input event denotes a value to be defined by the environment: any value is acceptable by the process.
– A symbolic variable in an output event denotes a value to be defined by the process. If nondeterminism means that there are several values that satisfy the constraint on it, any of those values can be chosen, internally. This means that, even from a stable configuration as defined above, there may still be open nondeterministic choices for the process.

The examples in the sequel illustrate how symbolic acceptances can be used.

Example 7 Internal and external choices We first consider an example of an internal choice involving an input and an output, where the output value is arbitrary: it is left completely unspecified in the model.

$$\begin{array}{l} \textbf{process } \mathit{IC} \mathrel{\widehat=} \textbf{begin} \\ \quad \textbf{state } S \mathrel{\widehat=} [\, x : \mathbb{N} \,] \bullet g?y \rightarrow \textbf{Stop} \sqcap h!x \rightarrow \textbf{Stop} \\ \textbf{end} \end{array}$$

In the *CashMachine*, the definition of *CardV* uses a similar kind of internal choice, since the recursive call to *CardV*, which starts with an input on *incard*, is in choice with, for example, an output communication on *disp*.

The set $csacceptances(\mathtt{IC}, (\langle\,\rangle, \mathtt{true}))$ contains all sets of symbolic events that include *both* $(\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N})$ and $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$. To obtain a concrete acceptance for IC after the empty trace, we need one instance of the input event $(\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N})$, but all the instances of the output event $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$.

In contraposition, we also consider the process EC below, which provides an external choice between an input and an output. In the *CashMachine* example, such a choice is offered by the action *CashC* (since semantically a synchronisation is a special form of output).

$$\begin{array}{l}\textbf{process}\ EC \mathrel{\widehat=} \textbf{begin}\\ \quad\textbf{state}\ S \mathrel{\widehat=} [\,x : \mathbb{N}\,] \bullet g?y \to \textbf{Stop} \ \Box\ h!x \to \textbf{Stop}\\ \textbf{end}\end{array}$$

Now, the acceptances are all the sets of symbolic events that include *either* $(\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N})$ or $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$. In a concrete acceptance, we can include either one instance of $(\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N})$ or all the instances of $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$. □

As these examples suggest, given a set of constrained symbolic acceptances, we need to consider the input and output events separately. For a set SX we define $SX\!\upharpoonright_I$ to be the subset of SX containing the input events.

$$SX\!\upharpoonright_I = \{\mathtt{d}, \alpha_0, \mathtt{c} \mid (\mathtt{d}?\alpha_0, \mathtt{c}) \in SX \bullet (\mathtt{d}.\alpha_0, \mathtt{c})\}$$

Similarly, $SX\!\upharpoonright_O$ contains the subset of events on output variables.

Another issue illustrated below is the treatment of multiple input or output events in the same set of symbolic acceptances.

Example 8 Multiple inputs and outputs We first consider an example of an internal choice involving inputs. External choices do not give rise to acceptance sets that need to have more than one event (unless, like in the *Overlap* example, they offer two choices of the same event, in which case it is really an internal choice written using the external choice operator).

$$\textbf{process}\ II \mathrel{\widehat=} \textbf{begin}\ \bullet g?x \to \textbf{Stop} \sqcap h?y \to \textbf{Stop}\ \textbf{end}$$

The set $csacceptances(\mathtt{II}, (\langle\,\rangle, \mathtt{true}))$ contains all sets that include *both* the event $(\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N})$ and the event $(\mathtt{h}?\alpha_0, \alpha_0 \in \mathbb{N})$. A concrete acceptance for II needs *one* instance of each of them.

In contraposition, we also consider the process IO below which makes an internal choice between outputs. The definition of *CardV* in the *CashMachine* has such a choice: between a synchronisation on $outcard!c$ (or rather, an output of the value c through *outcard*) and an output (of a) through *disp*.

$$\begin{array}{l}\textbf{process}\ IO \mathrel{\widehat=} \textbf{begin}\\ \quad\textbf{state}\ S \mathrel{\widehat=} [\,x : \mathbb{N}\,] \bullet g!x \to \textbf{Stop} \sqcap h!x \to \textbf{Stop}\\ \textbf{end}\end{array}$$

In this case, the constrained symbolic acceptances contains all the sets that include *both* $(\mathtt{g}!\alpha_0, \alpha_0 \in \mathbb{N})$ and $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$. A concrete acceptance includes *all* instances of $(\mathtt{g}!\alpha_0, \alpha_0 \in \mathbb{N})$ and *all* instances of $(\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})$. (There is only one minimal acceptance set.) □

In summary, if we take the sets of instances of the symbolic events in a set SX, to get a concrete acceptance set, we pick one instance from each of the sets defined by an input event, and unite all the sets coming from output events. This is formalised below by the function instAS from sets of input-output constrained symbolic events to sets of sets of concrete events.

Definition 8

$$\mathrm{instAS}(SX) = \{\, A : \otimes(\mathrm{instE} \,(\!|\, SX \restriction_I \,|\!)) \bullet A \cup \bigcup \mathrm{instE} \,(\!|\, SX \restriction_O \,|\!) \,\}$$

□

We consider separately the subsets of symbolic events representing inputs and outputs. We need to choose an element of each of the sets obtained from $SX \restriction_I$. The $\otimes$ operator defined below is a generalised cartesian product whose elements are sets, rather than tuples. It takes a set of sets SS as argument, and defines also a set of sets, characterised as follows.

$$\otimes SS = \left\{ S_1 \mid \begin{array}{l} (\forall\, S_2 : SS \mid S_2 \neq \varnothing \bullet (\exists\, e : S_2 \bullet e \in S_1)) \\ \wedge \\ (\forall\, e : S_1 \bullet (\exists\, S_2 : SS \bullet e \in S_2)) \end{array} \right\}$$

In the definition of $\mathrm{instAS}(SX)$, the $\otimes$ operator is used to determine the set of all sets of events that can be obtained by choosing one element in each of the sets $\mathrm{instE}\,(\!|\, SX \restriction_I \,|\!)$. The sets A so obtained can be used to form a concrete acceptance. What we need to add to such an A is the union of all sets of sets of events that are obtained from $SX \restriction_O$, namely $\bigcup \mathrm{instE} \,(\!|\, SX \restriction_O \,|\!)$.

Below, we describe how symbolic and concrete acceptances are related.

Proposition 4

$$\begin{array}{l} acceptances(\mathtt{P}, t) = \bigcup \{\, \mathtt{st}, \mathtt{c}, SX \mid \\ \qquad \left(\begin{array}{l} (\mathtt{st}, \mathtt{c}) \in cstraces^{\mathsf{a}}(\mathtt{P}) \wedge t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}) \wedge \\ SX \in csacceptances^{\mathsf{a}}(\mathtt{P}, (\mathtt{st}, \mathtt{c})) \end{array} \right) \\ \qquad \bullet\ \mathrm{instAS}((\!|SX|\!) \restriction^{\mathsf{a}} t) \\ \qquad \} \end{array}$$

□

We consider the constrained symbolic traces $(\mathtt{st}, \mathtt{c})$ that can be instantiated to the trace of interest t, and use them to characterise the symbolic acceptances SX of interest. Each such set gives rise to a set of concrete acceptances: a set of sets of events. Their union is the complete set of acceptances.

With $(\!|\, SX \,|\!) \restriction^{\mathsf{a}} t$, we restrict the events to the trace t: using relational image, we restrict each symbolic event in SX to consider the constraint determined by t. The resulting set of events is then instantiated using instAS.

Example 9 Overlap To calculate $acceptances(Overlap, \langle g.4 \rangle)$, we observe that there is a single symbolic trace of *Overlap* that can be instantiated to $\langle g.4 \rangle$; this is $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$. As discussed previously, a set of symbolic acceptances of *Overlap* after $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$ is as follows.

$$\{(\mathtt{g!}\alpha_1, \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0), (\mathtt{h!}\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0)\}$$

If we restrict these events to the trace $\langle g.4 \rangle$, we obtain the set below. Since

α_0 is now defined to be 4, the constraint on $\mathtt{h}.\alpha_1$ becomes *false*.

$$\{\, (\mathtt{g}!\alpha_1, \alpha_0 = \mathtt{4} \wedge \alpha_1 = \alpha_0), (\mathtt{h}!\alpha_1, \mathtt{false}) \,\}$$

When we instantiate this set using instAS, we observe that there are no events representing inputs: $SX \upharpoonright_I$ is empty, and so is instE $(\!|\, SX \upharpoonright_I \,|\!)$. Therefore, the only set A in $\otimes$(instE $(\!|\, SX \upharpoonright_I \,|\!)$) is the empty set, which does not contribute any event to the concrete acceptance sets. The set $SX \upharpoonright_O$, on the other hand, contains all the events of SX. Their instantiation using instE gives $\{\, \{g.4\}, \varnothing \,\}$, since the event on h has a *false* constraint. The distributed union of this set is $\{g.4\}$, which is indeed in $acceptances(Overlap, \langle g.4 \rangle)$. □

Example 10 Overlap In the case of $acceptances(Overlap, \langle g.6 \rangle)$, we observe that there are two constrained symbolic traces of *Overlap* that can be instantiated to $\langle g.6 \rangle$; they are $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$ and $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{5})$. If we restrict to $\langle g.6 \rangle$ the events of the set of constrained symbolic acceptances of *Overlap* after $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 < \mathtt{10})$ presented in Example 9, we get the result below.

$$\{\, (\mathtt{g}!\alpha_1, \alpha_0 = \mathtt{6} \wedge \alpha_1 = \alpha_0), (\mathtt{h}!\alpha_1, \alpha_0 = \mathtt{6} \wedge \alpha_1 = \alpha_0) \,\} \tag{11}$$

Its instantiation using instE gives $\{\, \{g.6\}, \{h.6\} \,\}$, whose distributed union is $\{g.6, h.6\}$, which is indeed an acceptance set in $acceptances(Overlap, \langle g.6 \rangle)$.

A set of acceptances of *Overlap* after $(\langle \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{5})$ is as follows.

$$\{(\mathtt{g}!\alpha_1, \mathtt{5} < \alpha_0 < \mathtt{10} \wedge \alpha_1 = \alpha_0), (\mathtt{h}!\alpha_1, \mathtt{5} < \alpha_0 \wedge \alpha_1 = \alpha_0)\}$$

The restriction of these events to $\langle g.6 \rangle$ gives the same result presented above in (11), so it gives rise to no new concrete acceptances. □

In the next sections, we use the definitions of symbolic traces, initials, and acceptances to characterise symbolic tests and test sets.

4 Testing for traces refinement

Traces refinement is useful for reasoning about safety properties of a process. We write $P_1 \sqsubseteq_T P_2$ when the process P_1 is trace refined by the process P_2; informally, this ensures that P_2 does not engage in any interactions with the environment that are not allowed by P_1. More precisely, this corresponds to trace inclusion: the set of traces of P_2 is included in that of P_1. In [16], we use the *Circus* UTP theory and its relationship with the traces (component of the failures-divergences) model of CSP, as identified in [18], to calculate a formal definition of traces refinement for *Circus* using its UTP model.

The function T_T below characterises tests for traces refinement as *Circus* actions. It takes as arguments a trace and a forbidden continuation e.

Definition 9

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

□

The extra events *pass*, *fail*, and *inc* are assumed not to be used in the given trace or in the forbidden continuation e, or more generally, in the reference *Circus* specification SP used to identify the traces and forbidden continuations of interest. These events indicate the verdict. A corresponding *Circus* test process has no state and the main action as defined above by T_T.

An execution $Execution_{SUT}^{SP}(T)$ of a test T for a given SUT, against a specification SP, is defined below as a process: the parallel execution of T with SUT, where all the events are hidden, except for *inc*, *pass*, and *fail*.

Definition 10 $Execution_{SUT}^{SP}(T) = (SUT \llbracket \Sigma \rrbracket T) \setminus \Sigma \quad \Box$

We use Σ to denote the set of channels in scope: those used in either the specification or in the SUT. (Like in CSP, we do not attach alphabets of events to processes, but to the parallelism operator, as said in Section 2.1.)

The test characterised by $T_T(t, e)$ drives the SUT to synchronise on the events of t. If it does not, this is not a mistake, since the SUT does not need to exhibit all the allowed behaviours of the specification. The test, however, is not conclusive in this case. If the trace t is accepted, then the SUT should not synchronise on the forbidden continuation e accepted by the test. If it does, then the test fails. Accordingly, if the last event of an execution is *pass*, then the SUT passes the test; similarly, if it is *fail*, then the SUT fails the test. Finally, if the last event is *inc*, then the test is inconclusive. As defined above, the synchronisation set, which is the interface of the system as defined in the specification SP, is hidden. This means that synchronisation between the SUT and the test proceeds immediately when available, and cannot be affected by the test execution environment.

We define an exhaustive test set for traces refinement based on the set of traces t of the reference specification SP, and their forbidden continuations: events e not in the set $initials(SP, t)$ containing all the events that may be accepted by SP after engaging in the trace t. The exhaustive test set $Exhaust_T(SP)$ for a process SP is a set of processes whose main actions are the tests in the set $ExhaustA_T(SP)$ containing the actions that result from applying T_T to all traces of SP and their forbidden continuations.

Definition 11

$$Exhaust_T(\mathtt{SP}) = \{\, TA : ExhaustA_T(\mathtt{SP}) \bullet (\mathtt{begin} \;\bullet\; \mathtt{TA}\ \mathtt{end}) \,\}$$

$$ExhaustA_T(\mathtt{SP}) = \{\, t, e \mid t \in traces(\mathtt{SP}) \land e \notin initials(\mathtt{SP}, t) \bullet T_T(t, e) \,\}$$

$\Box$

Exhaustivity is established by the following theorem. So far, all our definitions are very similar to those presented in [14] for CSP, and so the proof of this theorem is not surprising, and is basically omitted. For a trace (or any sequence) t, we use $last\ t$ to identify its last element.

Theorem 1 Exhaustivity - traces refinement *Given two Circus processes, SP and SUT, we have that $SP \sqsubseteq_T SUT$ if, and only if,*

$$\forall\, T : Exhaust_T(\mathtt{SP});\ t : traces(Execution_{SUT}^{SP}(T)) \bullet last\ t \neq fail$$

Proof Straightforward adaptation of that presented in [14] for a similar result for testing based on CSP specifications. $\square$

In this paper, we consider symbolic descriptions of the tests. In particular, we define in this section a set of symbolic tests that describe all tests in $\mathit{Exhaust}_T(\mathit{SP})$. As previously discussed, it is useful to explore elaborate selection criteria that take advantage of the specifications of the data operations and of the constraints of the symbolic traces and initials.

For a symbolic trace $(\mathtt{st}, \mathtt{c}_1)$ and a symbolic event $(\mathtt{d}.\alpha_0, \mathtt{c}_2)$ that represents a forbidden continuation, the corresponding symbolic test is given by $\mathtt{ST}^{\alpha}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2))$ defined below; it incorporates $\mathtt{c}_1$ and $\mathtt{c}_2$. For the prefixing involving $\mathtt{d}.\alpha_0$, we use $\mathtt{c}_2$ to restrict the accepted values of the symbolic variable. For the prefixings corresponding to each of the symbolic events $\mathtt{d}_{\mathtt{st}}.\alpha_0$ of the trace $\mathtt{st}$, we use the constraint $\mathtt{c}_1$ to extract the right constraint over 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 $\overline{\alpha, \alpha_0}$.

Definition 12

$$\mathtt{ST}^{\alpha}_{\mathtt{T}}((\langle\,\rangle, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2)) = \mathtt{pass} \to \mathtt{d}?\alpha_0 : \mathtt{c}_2 \to \mathtt{fail} \to \mathtt{Stop}$$

$$\begin{array}{l}\mathtt{ST}^{\alpha}_{\mathtt{T}}((\langle\, \mathtt{d}_{\mathtt{st}}.\alpha_0 \,\rangle \frown \mathtt{st}, \mathtt{c}), \mathtt{iose}) = \\ \quad \mathtt{inc} \to \mathtt{d}_{\mathtt{st}}?\alpha_0 : (\exists\, \overline{\alpha, \alpha_0} \bullet \mathtt{c}) \to \mathtt{ST}^{(\alpha,\alpha_0)}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}), \mathtt{iose})\end{array}$$

$\square$

Example 11 NoOverlap For the constrained symbolic trace $(\langle\, \mathtt{g}.\alpha_0 \rangle, \alpha_0 > \mathtt{10})$ of *NoOverlap*, and its forbidden continuation $(\mathtt{g}.\alpha_1, \alpha_0 > \mathtt{10} \wedge \alpha_1 \neq \alpha_0)$, as characterised by $\overline{\mathit{csinitials}}$, we have the test below.

$$\begin{array}{l}\mathtt{inc} \to \mathtt{g}?\alpha_0 : (\alpha_0 > \mathtt{10}) \\ \to \mathtt{pass} \to \mathtt{g}?\alpha_1 : (\alpha_0 > \mathtt{10} \wedge \alpha_1 \neq \alpha_0) \\ \to \mathtt{fail} \to \mathtt{Stop}\end{array}$$

The constraint associated with $\mathtt{g}?\alpha_0$ is on a single variable α_0, which is not quantified. The constraint on the $\mathtt{g}?\alpha_1$ relates the fresh variable α_1 to the previously communicated value represented by α_0. $\square$

Example 12 CashMachine As suggested by Example 18, one of the constrained symbolic traces of *CashMachine* is as follows.

$$\left(\begin{array}{l}\langle\, \mathtt{refill}, \mathtt{incard}.\alpha_0.\alpha_1.\alpha_2, \mathtt{outcard}.\alpha_3 \rangle, \\ \alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\, \alpha_0 \wedge \alpha_2 \in \mathbb{N}_1 \wedge \alpha_3 = \alpha_0\end{array}\right)$$

In addition, a forbidden continuation of this trace has the symbolic event $\mathtt{outcard}.\alpha_4$ and the same constraint. From these, we get the test below.

$$\begin{array}{l}\mathtt{inc} \to \mathtt{refill} \\ \to \mathtt{inc} \to \mathtt{incard}?\alpha_0?\alpha_1?\alpha_2 : (\alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\, \alpha_0 \wedge \alpha_2 \in \mathbb{N}_1) \\ \to \mathtt{inc} \to \mathtt{outcard}?\alpha_3 : (\alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\, \alpha_0 \wedge \alpha_2 \in \mathbb{N}_1 \wedge \alpha_3 = \alpha_0) \\ \to \mathtt{pass} \to \mathtt{outcard}?\alpha_4 : (\alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\, \alpha_0 \wedge \alpha_2 \in \mathbb{N}_1 \wedge \alpha_3 = \alpha_0) \\ \to \mathtt{fail} \to \mathbf{Stop}\end{array}$$

With this we check that it is not possible to output a card twice. $\square$

These symbolic tests are in fact proper *Circus* processes or actions. As tests, however, they provide little controllability. The inputs $\mathtt{d}?\alpha_0$ that correspond to inputs of the *SUT* are chosen nondeterministically in a test execution, since d is hidden. In practice, the symbolic tests are not meant to be used directly; they define patterns for the tests to be constructed by instantiation. The instances provide particular values to be communicated.

An instance of a symbolic test is defined by the functions instP and instA, which specify the set of test instances of a symbolic process or action test. We define instA here just for the kind of actions that can be used to characterise tests for trace refinement. In the next section, we extend its definition to consider external choices, which are used in the tests for *conf*.

Definition 13

$$\mathrm{instP}(\mathtt{begin} \bullet \mathtt{STA}\ \mathtt{end}) = \{\, TA : \mathrm{instA}(STA) \bullet (\mathbf{begin} \bullet TA\ \ \mathbf{end}) \,\}$$

$$\begin{array}{l}
\mathrm{instA}(\mathtt{Stop}) = \{\, \mathbf{Stop} \,\} \\
\mathrm{instA}(\mathtt{d} \to \mathtt{A}) = \{\, TA : \mathrm{instA}(A) \bullet d \to TA \,\} \\
\mathrm{instA}(\mathtt{d}?\alpha_0 : \mathtt{c} \to \mathtt{A}) = \\
\quad \{\, v_0, TA \mid c[v_0/\alpha_0] \land TA \in \mathrm{instA}(\mathtt{A}[\mathtt{v}_0/\alpha_0]) \bullet d.v_0 \to TA \,\}
\end{array}$$

□

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

We observe that in the symbolic test $\mathtt{d}?\alpha_0 : \mathtt{c} \to \mathtt{A}$ the symbolic variable is being used as a local input variable whose scope is `A`. We, therefore, use the standard notation $\mathtt{A}[\mathtt{v}_0/\alpha_0]$ to denote substitution in the action `A` of $\mathtt{v}_0$ for its free variable α_0, as defined for *Circus* [52].

The next lemma establishes an important property of the function instA, when applied to symbolic tests for actions. Namely, given a symbolic test $\mathtt{ST}^{\alpha}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2))$, by choosing values v for the symbolic variables in use, that is, those in the sequence α, and then instantiating the resulting symbolic test, we obtain the same set of concrete tests that result if we record the choice of values v for α in the trace $(\mathtt{st}, \mathtt{c}_1)$ and in the event $(\mathtt{d}.\alpha_0, \mathtt{c}_2)$, and then instantiate the symbolic test obtained from the resulting symbolic trace and event. We use α to stand for a sequence of symbolic variables, but in the provisos write α also to denote the set of the elements in the sequence.

Lemma 5 Alphabet of instantiations - traces-refinement tests

$$\begin{array}{l}
\mathrm{instA}((\mathtt{ST}^{\alpha}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2)))[\mathtt{v}/\alpha]) = \\
\quad \mathrm{instA}(\mathtt{ST}^{\langle\rangle}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1[\mathtt{v}/\alpha]), (\mathtt{d}.\alpha_0, \mathtt{c}_2[\mathtt{v}/\alpha])))
\end{array}$$

provided $\alpha_0 \notin \alpha$, *and* $\alpha \cap \alpha\mathtt{st} = \varnothing$. □

The following set of symbolic tests represents the exhaustive test set. Just like

$Exhaust_T(SP)$ is defined in terms of the sets of concrete traces and initials of SP, $\mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP})$ is defined in terms of the constrained symbolic traces and initials of SP. The alphabet $\mathtt{a}$ used is fixed by an extra parameter.

Definition 14

$$\mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) = \{\, \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet (\mathtt{begin} \;\bullet \mathtt{STA}\ \mathtt{end}) \,\}$$

$$\mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) = \left\{ \begin{array}{l} \mathtt{cst}, \mathtt{cse} \mid \mathtt{cst} \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge \mathtt{cse} \in \overline{\mathit{csinitials}}^{\mathtt{a}}(\mathtt{SP}, \mathtt{cst}) \\ \bullet\ \mathtt{ST}^{\langle\rangle}_{\mathtt{T}}(\mathtt{cst}, \mathtt{cse}) \end{array} \right\}$$

□

In order to prove exhaustivity, first, we relate a symbolic test for actions to a set of concrete tests. Namely, the instances of a symbolic test are the tests obtained from the instances of the symbolic trace and event used to construct it. The pairs of symbolic trace $(\mathtt{st}, \mathtt{c}_1)$ and event $(\mathtt{d}.\alpha_0, \mathtt{c}_2)$ considered are such that $c_2 \Rightarrow c_1$ so that we know that $(\mathtt{d}.\alpha_0, \mathtt{c}_2)$ is feasible after $(\mathtt{st}, \mathtt{c}_1)$. This holds for all events in $\mathit{csinitials}^{\mathtt{a}}(\mathtt{P}, (\mathtt{stc}))$ with respect to $(\mathtt{st}, \mathtt{c})$.

Lemma 6 Instantiation of symbolic tests-actions-traces refinement

$$\begin{array}{l} \mathrm{instA}(\mathtt{ST}^{\langle\rangle}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2))) = \\ \qquad \{\, t, e \mid t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_1) \wedge e \in \mathrm{instE}^{\alpha\mathtt{st}}((\mathtt{d}.\alpha_0, \mathtt{c}_2) \upharpoonright^{\alpha\mathtt{st}} t) \bullet T_T(t, e) \,\} \end{array}$$

provided $\alpha\mathtt{c}_2 \subseteq \alpha\mathtt{st} \cup \{\alpha_0\}$*,* $\alpha_0 \notin \alpha\mathtt{st}$*, and* $c_2 \Rightarrow c_1$*.* □

Now, we relate the symbolic and concrete exhaustive test sets for actions. This next lemma is the central result in the argument summarised in Theorem 2 below that justifies the use of the set $\mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{P})$ of symbolic tests to characterise the exhaustive test set for traces refinement.

Lemma 7 Symbolic exhaustivity - actions - traces refinement *For every alphabet a*

$$ExhaustA_T(\mathtt{SP}) = \bigcup \{\, \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet \mathrm{instA}(\mathtt{STA}) \,\}$$

Proof

$$\bigcup \{\, \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet \mathrm{instA}(\mathtt{STA}) \,\}$$

$$= \bigcup \left\{ \begin{array}{l} \mathtt{cst}, \mathtt{cse} \mid \\ \quad \mathtt{cst} \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge \mathtt{cse} \in \overline{\mathit{csinitials}}^{\mathtt{a}}(\mathtt{SP}, \mathtt{cst}) \\ \bullet\ \mathrm{instA}(\mathtt{ST}^{\langle\rangle}_{\mathtt{T}}(\mathtt{cst}, \mathtt{cse})) \end{array} \right\}$$

[definition of $\mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}$]

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}_1, \mathtt{d}, \alpha_0, \mathtt{c}_2 \mid \\ \quad (\mathtt{st}, \mathtt{c}_1) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge (\mathtt{d}.\alpha_0, \mathtt{c}_2) \in \overline{\mathit{csinitials}}^{\mathtt{a}}(\mathtt{SP}, (\mathtt{st}, \mathtt{c}_1)) \\ \bullet\ \mathrm{instA}(\mathtt{ST}^{\langle\rangle}_{\mathtt{T}}((\mathtt{st}, \mathtt{c}_1), (\mathtt{d}.\alpha_0, \mathtt{c}_2))) \end{array} \right\}$$

[property of sets]

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}_1, \mathtt{d}, \alpha_0, \mathtt{c}_2 \mid \\ \quad \left(\begin{array}{l} (\mathtt{st}, \mathtt{c}_1) \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{SP}) \wedge \\ (\mathtt{d}.\alpha_0, \mathtt{c}_2) \in \overline{\mathit{csinitials}}^{\mathsf{a}}(\mathtt{SP}, (\mathtt{st}, \mathtt{c}_1)) \end{array} \right) \\ \bullet \left\{ \begin{array}{l} t, e \mid t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_1) \wedge e \in \mathrm{instE}^{\alpha\mathtt{st}}((\mathtt{d}.\alpha_0, \mathtt{c}_2) \restriction^{\alpha\mathtt{st}} t) \\ \bullet\ T_T(t, e) \end{array} \right\} \end{array} \right\}$$

[Lemma 6 and properties of *cstraces* and $\overline{\mathit{csinitials}}$]

$$= \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}_1, \mathtt{d}, \alpha_0, \mathtt{c}_2, t, e \mid \\ \quad \left(\begin{array}{l} (\mathtt{st}, \mathtt{c}_1) \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{SP}) \wedge \\ (\mathtt{d}.\alpha_0, \mathtt{c}_2) \in \overline{\mathit{csinitials}}^{\mathsf{a}}(\mathtt{SP}, (\mathtt{st}, \mathtt{c}_1)) \wedge \\ t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_1) \wedge e \in \mathrm{instE}^{\alpha\mathtt{st}}((\mathtt{d}.\alpha_0, \mathtt{c}_2) \restriction^{\alpha\mathtt{st}} t) \end{array} \right) \\ \bullet\ T_T(t, e) \end{array} \right\}$$

[property of sets]

$$= \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}_1, t, e \mid \\ \quad (\mathtt{st}, \mathtt{c}_1) \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{SP}) \wedge \mathtt{t} \in \mathrm{instT}(\mathtt{st}, \mathtt{c}_1) \wedge e \notin \mathit{initials}(\mathtt{SP}, t) \\ \bullet\ T_T(t, e) \end{array} \right\}$$

[Lemma 4]

$= \{ t, e \mid t \in \mathit{traces}(\mathtt{SP}) \wedge e \notin \mathit{initials}(\mathtt{SP}, t) \bullet T_T(t, e) \}$ [Proposition 1]

$= \mathit{ExhaustA}_T(\mathtt{SP})$ [definition of $\mathit{ExhaustA}_T$]

□

Finally, we can lift our result on symbolic action tests to process tests.

Theorem 2 Symbolic exhaustivity-processes-traces refinement *For every alphabet* a*,*

$$\mathit{Exhaust}_T(\mathtt{SP}) = \bigcup \{ \mathtt{ST} : \mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet \mathrm{instP}(\mathtt{ST}) \}$$

Proof

$\bigcup \{ \mathtt{ST} : \mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet \mathrm{instP}(\mathtt{ST}) \}$

$= \bigcup \{ \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP}) \bullet \mathrm{instP}(\mathtt{begin} \bullet \mathtt{STA\ end}) \}$

[definition of $\mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{T}}$]

$= \{ \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{T}}(\mathtt{SP});\ \mathit{TA} : \mathrm{instA}(\mathtt{STA}) \bullet \mathbf{begin} \bullet \mathit{TA}\ \mathbf{end} \}$

[definition of instP]

$= \{ \mathit{TA} : \mathit{ExhaustA}_T(\mathit{SP}) \bullet \mathbf{begin} \bullet \mathit{TA}\ \mathbf{end} \}$ [Lemma 7]

$= \mathit{Exhaust}_T(\mathtt{SP})$ [definition of $\mathit{Exhaust}_T$]

□

This justifies the soundness of using the symbolic tests as a basis for selection

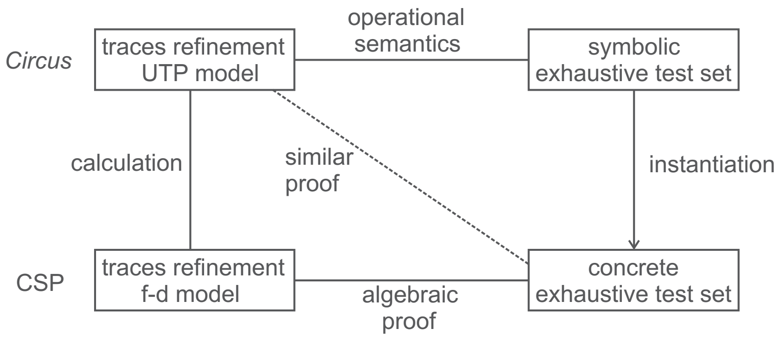
Circus
traces refinement UTP model
operational semantics
symbolic exhaustive test set
calculation
similar proof
instantiation
CSP
traces refinement f-d model
algebraic proof
concrete exhaustive test set

Fig. 3 Structure of the argument of exhaustivity

of concrete tests. It guarantees that, by working with the symbolic representation of tests, we are not excluding any of the possible concrete tests, or including unnecessary tests. Figure 3 summarises the structure of our argument. In previous work, based on CSP and its failures-divergences model, we have established exhaustivity of a set of concrete tests. Using a relationship between the failures-divergences model and the UTP model for CSP and *Circus*, we have calculated a notion of traces refinement for *Circus*. The correspondence between the notions of traces refinement and the fact that the concrete tests induced by *Circus* models are similar to those obtained using CSP allowed us to have a very similar proof of exhaustivity in the context of *Circus* (Theorem 1). The operational semantics of *Circus* induces a symbolic characterisation of the tests, which we proved that, by instantiation as formalised in Definition 13, can be used to define the exhaustive set of concrete tests (Theorem 2). The same proof approach is used in our argument of exhaustivity for *conf* in the next section.

Ultimately, the predicative specifications used in *Circus* to define data types and operations in the style of Z specify sets of values that characterise the extensional meaning of these types and operations. Theorem 1 is justified by a model based on sets of values. The purpose of establishing the connection to this result via Theorem 2 is to establish that our predicative model of tests based on the symbolic operational semantics of *Circus* is sound. The predicative model supports and guides the use of more advanced reasoning and testing techniques (based, for example, on SAT and SMT solving).

5 Testing for *conf*

The *conf* relation captures reduction of deadlock; it is a widely used notion of conformance in testing. We write P_1 *conf* P_2 if, and only if, whenever P_2 engages in a sequence of events that can be accepted by P_1, then P_2 can only deadlock if P_1 may as well. Formally, *conf* can be defined as follows.

$$P_2 \ \mathit{conf}\ P_1 \mathrel{\widehat=} \forall\, t : \mathit{traces}(P_1) \cap \mathit{traces}(P_2) \bullet \mathit{Ref}(P_2, t) \subseteq \mathit{Ref}(P_1, t)$$
$$\text{where } \mathit{Ref}(P, t) \mathrel{\widehat=} \{\, X \mid (t, X) \in \mathit{failures}(P) \,\}$$

We note that the definition of $\mathit{Ref}(P, t)$ is compatible with that of $\mathit{refusals}(P)$

in CSP, for the process P/t [58, pages 94,197]. A formalisation of *conf* is calculated for the *Circus* denotational UTP model in [16]. Like that of traces refinement, it is based on the above definition and a relationship between the *Circus* UTP theory and the failures-divergences model [18].

For SUT *conf* SP to hold, the definition requires that, after performing every one of their common traces, the failures of the SUT are failures of SP. This means that, after a trace t of SP, the SUT cannot refuse all events in an acceptance set X of SP. It also means that the SUT may refuse all events refused by SP or accept some of them. Testing for *conf* based on the refusals of SP is, therefore, useless. Instead, we execute the test corresponding to t, followed by an external choice among the events in X.

In more detail, for a trace t, and a set of acceptances X, a test is characterised by $T_F(t, X)$ as defined below. It offers the trace t, and at the end a choice of the events in X. If the SUT does not synchronise on all events of t, the test is inconclusive, since it is not necessary for the SUT to allow for all the traces indicated in the specification. If the trace t is accepted, however, then the SUT must accept at least one event of X.

Definition 15

$$T_F(\langle\rangle, X) = \mathit{fail} \rightarrow (\Box\ e : X \bullet e \rightarrow \mathit{pass} \rightarrow \mathbf{Stop})$$
$$T_F(\langle e \rangle \frown t, X) = \mathit{inc} \rightarrow e \rightarrow T_F(t, X)$$

□

The exhaustive test set for *conf* contains all the tests obtained from the traces and acceptances of the *Circus* specification SP.

Definition 16

$$\mathit{Exhaust}_F(\mathtt{SP}) = \{\ \mathit{TA} : \mathit{ExhaustA}_F(\mathtt{SP}) \bullet (\mathbf{begin}\ \bullet\ \mathit{TA}\ \ \mathbf{end})\ \}$$
$$\mathit{ExhaustA}_F(\mathtt{SP}) =$$
$$\{\ t, e \mid t \in \mathit{traces}(\mathtt{SP}) \land X \in \mathit{acceptances}(\mathtt{SP}, t) \bullet T_F(t, X)\ \}$$

□

The sets of minimal acceptances are actually enough, but we leave aside the technicalities of the removal of the unnecessary tests. With that, we have a clearer structure of definitions and proofs.

Theorem 3 Exhaustivity - *conf* *Given two Circus processes, SP and SUT, we have that SUT conf SP if, and only if,*

$$\forall\ T : \mathit{Exhaust}_F(\mathtt{SP});\ t, X \mid (t, X) \in \mathit{failures}(\mathit{Execution}_{SUT}^{SP}(T)) \bullet$$
$$\mathit{last}\ \ t \neq \mathit{fail} \lor X \neq \{\ \mathit{inc}, \mathit{pass}, \mathit{fail}\ \}$$

Proof Straightforward simplification of the proof presented in [14] for CSP. Here we do not need to justify the use of minimal acceptance sets, since this issue will be treated in subsequent work as an optimisation. □

In the context of *Circus*, our interest is of course the definition of symbolic tests, based on the sets of constrained symbolic traces and acceptances. They are defined by the function $\mathtt{ST}_\mathtt{F}^\alpha$ defined in the sequel. As for $\mathtt{ST}_\mathtt{T}^\alpha$, the α parameter records the symbolic variables already used.

Definition 17

$$\mathtt{ST}_\mathtt{F}^\alpha((\langle\,\rangle, \mathtt{c}), SX) =$$

$$\mathtt{fail} \to \left(\begin{array}{l} \Box (\mathtt{d}.\alpha_0, \mathtt{c}) : SX\upharpoonright_I \bullet \mathtt{d}?\alpha_0 : \mathtt{c} \to \mathtt{pass} \to \mathtt{Stop} \\ \Box \\ \Box (\mathtt{d}.\alpha_0, \mathtt{c}) : SX\upharpoonright_O \bullet \mathtt{d}!\alpha_0 : \mathtt{c} \to \mathtt{pass} \to \mathtt{Stop} \end{array}\right)$$

$$\mathtt{ST}_\mathtt{F}^\alpha((\langle\,\mathtt{d}.\alpha_0\,\rangle \frown \mathtt{st}, \mathtt{c}), SX) =$$
$$\mathtt{inc} \to \mathtt{d}?\alpha_0 : (\exists\, \overline{\alpha, \alpha_0} \bullet \mathtt{c}) \to \mathtt{ST}_\mathtt{F}^{(\alpha,\alpha_0)}((\mathtt{st}, \mathtt{c}), SX)$$

□

Example 13 Symbolic test for IC For the process *IC*, we consider the trace $(\langle\,\rangle, \mathtt{true})$, and the symbolic acceptance set $\{\, (\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N}), (\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})\,\}$. With $\mathtt{ST}_\mathtt{F}^{\langle\rangle}((\langle\,\rangle, \mathtt{true}), \{\, (\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N}), (\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N})\,\})$ we get the test shown below. Since the leading trace is empty, the first event is *fail*.

$$\mathtt{fail} \to \left(\begin{array}{l} \mathtt{g}?\alpha_0 : (\alpha_0 \in \mathbb{N}) \to \mathtt{pass} \to \mathtt{Stop} \\ \Box \\ \mathtt{h}!\alpha_0 : (\alpha_0 \in \mathbb{N}) \to \mathtt{pass} \to \mathtt{Stop} \end{array}\right)$$

Apart from the annotation of the constraint on the output, this is a proper *Circus* action. Most importantly, however, it characterises a set of concrete tests; instantiation is discussed in the sequel. □

Example 14 Symbolic test for the CashMachine For the constrained symbolic trace $(\langle\,\mathtt{refill}, \mathtt{incard}.\alpha_0.\alpha_1.\alpha_2\rangle, \alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\,\alpha_0 \wedge \alpha_2 \in \mathbb{N}_1)$ of the *CashMachine* process, we have the symbolic acceptance below.

$$\left\{\begin{array}{l} \left(\begin{array}{l} \mathtt{incard}?\alpha_3!\alpha_4?\alpha_5, \\ \alpha_0 \in \mathtt{CARD} \wedge \ldots \wedge \alpha_3 \in \mathtt{CARD} \wedge \alpha_4 = \mathtt{pin}\,\alpha_3 \wedge \alpha_5 \in \mathbb{N}_1, \end{array}\right), \\ (\mathtt{outcard}!\alpha_3, \alpha_0 \in \mathtt{CARD} \wedge \ldots \wedge \alpha_3 = \alpha_0), \\ (\mathtt{cash}!\alpha_3, \alpha_0 \in \mathtt{CARD} \wedge \ldots \wedge \Sigma\,\alpha_3 = \alpha_2 \wedge \forall\,\mathtt{n} : \mathtt{Note} \bullet \alpha_3 \sharp\, \mathtt{n} \leq \mathtt{cap}) \end{array}\right\}$$

This reflects the fact that, after inputting a card, we have to be prepared for the machine to produce no cash, retain the card, and then accept a new card, or return the card previously input, or finally produce the cash requested. The symbolic test corresponding to this trace and acceptance is as follows. It checks that at least one of these reactions is observed.

$$\begin{array}{l} \mathtt{inc} \to \mathtt{refill} \\ \to \mathtt{inc} \to \mathtt{incard}?\alpha_0?\alpha_1?\alpha_2 : (\alpha_0 \in \mathtt{CARD} \wedge \alpha_1 = \mathtt{pin}\,\alpha_0 \wedge \alpha_2 \in \mathbb{N}_1) \\ \to \mathtt{fail} \to \\ \quad \left(\begin{array}{l} \mathtt{incard}?\alpha_3!\alpha_4?\alpha_5 : (\alpha_0 \in \mathtt{CARD} \wedge \ldots \wedge \alpha_5 \in \mathbb{N}_1) \to \mathtt{pass} \to \mathtt{Stop} \\ \Box \\ \mathtt{outcard}!\alpha_3 : (\alpha_0 \in \mathtt{CARD} \wedge \ldots \wedge \alpha_3 = \alpha_0) \to \mathtt{pass} \to \mathtt{Stop} \\ \Box \\ \mathtt{cash}!\alpha_3 : (\ldots \wedge \forall\,\mathtt{n} : \mathtt{Note} \bullet \alpha_3 \sharp\, \mathtt{n} \leq \mathtt{cap}) \to \mathtt{pass} \to \mathtt{Stop} \end{array}\right) \end{array}$$

This illustrates that a (symbolic) event may involve both inputs and outputs;

this is the case of $\mathtt{incard}?\alpha_3!\alpha_4?\alpha_5$. Our definitions do not (explicitly) cover these examples; their extension, however, is long, but not difficult. □

The purpose of instantiation is to achieve controllability. There is no point in trying to control the output of a process; the choice is internal. So, in the instantiation of a test, we choose only values for the inputs. We formalise instantiation below, but first we present an example.

Example 15 Concrete tests for IC In the symbolic test defined above for IC, we have an external choice involving a communication over an input variable and another over an output variable. The instances of this test are as follows.

$$fail \to (g.0 \to pass \to \mathbf{Stop} \mathbin{\square} h?\alpha_0 : (\alpha_0 \in \mathbb{N}) \to pass \to \mathbf{Stop})$$
$$fail \to (g.1 \to pass \to \mathbf{Stop} \mathbin{\square} h?\alpha_0 : (\alpha_0 \in \mathbb{N}) \to pass \to \mathbf{Stop})$$
$$\ldots$$

In a test execution, the input for the SUT is fixed, but any output is accepted. □

Even if we instantiated the output communications, the external choices in the instantiated tests would need to include all possible instantiations, because it is not a failure for the SUT to refuse to output a particular value: it may output any of those that satisfy the (possibly nondeterministic) specification of the output. The input prefixing that we keep is just an abbreviated way of expressing this possibly infinite external choice; a similar approach was adopted in [43]. More precisely, we have the result below: a direct consequence of properties of prefixing and choice in *Circus* [54].

Lemma 8 Equivalence of *conf* tests

$$T_F(t, \{d.\alpha_0 : c\}) = T_F(t, \mathrm{instE}(\mathtt{d}.\alpha_0, \mathtt{c}))$$

□

In contraposition, in the tests for traces, and also in the part of the symbolic tests for *conf* that is determined by the trace rather than by the acceptance set, we instantiate output communications. As a result, we get inconclusive tests, when the SUT refuses to output a value chosen in a particular instantiated test. The verdict is not wrong, but there is an obvious scope for optimisation in the choice and construction of tests.

As with tests for traces refinement, instantiation of a symbolic test for *conf* is characterised by the functions instP and instA in Definition 13. The definition of instA for external choices is as follows.

$$\begin{array}{l}
\mathrm{instA}((\Box\, i \bullet \mathtt{d}_i?\alpha_i : \mathtt{c}_i \to \mathtt{A}(i)) \mathbin{\square} (\Box\, j \bullet \mathtt{d}_j!\alpha_j : \mathtt{c}_j \to \mathtt{A}(j))) = \\
\quad \Box\, (\!|\, \{\ STA : \otimes\{i \bullet \mathrm{instA}(\mathtt{d}_i?\alpha_i : \mathtt{c}_i \to \mathtt{A}(i))\ \} \\
\qquad\qquad \bullet STA \cup \{j, TA \mid TA \in \mathrm{instA}(\mathtt{A(j)}) \bullet d_j?\alpha_j : c_j \to TA\} \\
\qquad\quad \}\, |\!)
\end{array}$$

To create a concrete test from an external choice of prefixings, we need to handle the input and output prefixings separately. The result is a set of external choices of prefixings; they are obtained using a generalised $\Box$ operator that applies to sets of actions. It is applied to each of the sets in a set of sets of prefixings. From the set of sets of actions obtained by instantiating each of the input prefixings, we can form the set of sets of actions that should be included in the choice using the generalised cartesian product $\otimes$. It guarantees that the external choices are built out of sets of actions that contain one instance of each of the actions expressed using the symbolic events. To form the final sets of actions, we add to the sets the output prefixings, after turning them into inputs, and with the associated actions instantiated.

The following lemma is similar to Lemma 5; it establishes that partial trace instantiation can be carried out before or after the construction of a symbolic test also in the case of the *conf* tests defined above.

Lemma 9 Alphabet of instantiations - ***conf*** **tests**

$$\text{instA}((\mathtt{ST}^{\alpha}_{\mathtt{F}}((\mathtt{st}, \mathtt{c}), SX))[\mathtt{v}/\alpha]) = \text{instA}(\mathtt{ST}^{\langle\rangle}_{\mathtt{F}}((\mathtt{st}, \mathtt{c}[\mathtt{v}/\alpha]), SX[\mathtt{v}/\alpha]))$$

provided $\alpha \cap \alpha\mathtt{st} = \varnothing$. $\square$

The set of symbolic tests below represents the exhaustive set in Definition 16.

Definition 18

$$\mathtt{SExhaust}^{\mathtt{a}}_{\mathtt{F}}(\mathtt{SP}) = \{\, \mathtt{STA} : \mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{F}}(\mathtt{SP}) \bullet (\mathtt{begin} \ \bullet \mathtt{STA}\ \mathtt{end}) \,\}$$

$$\mathtt{SExhaustA}^{\mathtt{a}}_{\mathtt{F}}(\mathtt{SP}) = \left\{ \begin{array}{l} \mathtt{cst}, SX \mid \mathtt{cst} \in \mathit{cstraces}^{\mathsf{a}}(\mathtt{SP}) \wedge SX \in \mathit{csacceptances}^{\mathsf{a}}(\mathtt{SP}, \mathtt{cst}) \\ \bullet\ \mathtt{ST}^{\langle\rangle}_{\mathtt{F}}(\mathtt{cst}, \mathtt{SX}) \end{array} \right\}$$

$\square$

As for trace refinement, to relate the symbolic exhaustive test set with the concrete exhaustive test set for *conf*, we first examine the instantiation of individual symbolic tests. We compare the concrete tests obtained by instantiating a symbolic test, with those obtained from the concrete traces and acceptance sets obtained by instantiating the corresponding symbolic trace and acceptance set. For that, we define a function E that creates an input communication out of a constrained symbolic event.

Definition 19 $\text{E}(\mathtt{d}.\alpha_0, \mathtt{c}) = d?\alpha : c$ $\square$

We also define partial instantiation of a set of symbolic acceptances.

Definition 20

$$\text{pinstAS}(SX) = \{\, A : \otimes(\text{instE} \limits (\!| SX \upharpoonright_I |\!)) \bullet A \cup \text{E} (\!| SX \upharpoonright_O |\!) \,\}$$

$\square$

This definition is similar to that provided for instAS(SX) in Definition 8, but in the sets of pinstAS(SX) the output events are not instantiated; they are converted to proper *Circus* events as characterised by E.

Example 16 IC We consider again the set of symbolic acceptances presented in Example 13: $\{\, (\mathtt{g}?\alpha_0, \alpha_0 \in \mathbb{N}), (\mathtt{h}!\alpha_0, \alpha_0 \in \mathbb{N}) \,\}$ By taking the relational image of instE over the singleton $\{(\mathtt{g}.\alpha_0, \alpha_0 \in \mathbb{N})\}$, containing the only input event in the acceptance set, we get $\{\, \{g.0, g.1, \ldots\} \,\}$. With the $\otimes$ operator, we then get $\{\, \{g.0\}, \{g.1\}, \ldots\}$. On the other hand, the relational image of E over $\{(\mathtt{h}.\alpha_0, \alpha_0 \in \mathbb{N})\}$ gives just $\{h?\alpha_0 : (\alpha_0 \in \mathbb{N})\}$. Therefore, the result of applying pinstAS to the above set of acceptances gives the set $\{\, \{g.0, h?\alpha_0 : \alpha_0 \in \mathbb{N}\}, \{g.1, h?\alpha_0 : \alpha_0 \in \mathbb{N}\}, \ldots\}$. This is the basis for constructing the concrete tests in Example 15. □

Using the above notion of partial instantiation of a set of symbolic acceptances, we can relate symbolic and concrete tests as follows.

Lemma 10 Instantiation of symbolic tests - actions - *conf*

$$\begin{array}{l}\mathrm{instA}(\mathtt{ST}_{\mathtt{F}}^{\langle\rangle}((\mathtt{st},\mathtt{c}), SX)) = \\ \quad \{\, t, X \mid t \in \mathrm{instT}(\mathtt{st},\mathtt{c}) \wedge X \in \mathrm{pinstAS}((\!| SX |\!) \restriction^{\alpha\mathtt{st}} t) \bullet T_F(t, X) \,\}\end{array}$$

□

The relationship between the symbolic and concrete tests for actions is established by the lemma below. It is similar to Lemma 7.

Lemma 11 Symbolic exhaustivity - actions - conf *For every alphabet* $\mathtt{a}$*,*

$$ExhaustA_F(\mathtt{SP}) = \bigcup\{\, \mathtt{STA} : \mathtt{SExhaustA}_{\mathtt{T}}^{\mathtt{a}}(\mathtt{SP}) \bullet \mathrm{instA}(\mathtt{STA}) \,\}$$

Proof

$$\bigcup\{\, \mathtt{STA} : \mathtt{SExhaustA}_{\mathtt{F}}^{\mathtt{a}}(\mathtt{SP}) \bullet \mathrm{instA}(\mathtt{STA}) \,\}$$

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}, SX \mid \\ \quad (\mathtt{st},\mathtt{c}) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge SX \in \mathit{csacceptances}^{\mathtt{a}}(\mathtt{SP}, (\mathtt{st},\mathtt{c})) \\ \bullet\ \mathrm{instA}(\mathtt{ST}_{\mathtt{F}}^{\langle\rangle}((\mathtt{st},\mathtt{c}), SX)) \end{array} \right\}$$

[definition of $\mathtt{SExhaustA}_{\mathtt{T}}^{\mathtt{a}}$]

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}, SX \mid \\ \quad (\mathtt{st},\mathtt{c}) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge SX \in \mathit{csacceptances}^{\mathtt{a}}(\mathtt{SP}, (\mathtt{st},\mathtt{c})) \\ \bullet \left\{ \begin{array}{l} t, X \mid \\ \quad t \in \mathrm{instT}(\mathtt{st},\mathtt{c}) \wedge X \in \mathrm{pinstAS}((\!| SX |\!) \restriction^{\alpha\mathtt{st}} t) \\ \bullet\ T_F(t, X) \end{array} \right\} \end{array} \right\}$$

[Lemma 10]

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}, SX \mid \\ \quad (\mathtt{st},\mathtt{c}) \in \mathit{cstraces}^{\mathtt{a}}(\mathtt{SP}) \wedge \mathtt{SX} \in \mathit{csacceptances}^{\mathtt{a}}(\mathtt{SP}, (\mathtt{st},\mathtt{c})) \\ \bullet \left\{ \begin{array}{l} t, X \mid t \in \mathrm{instT}(\mathtt{st},\mathtt{c}) \wedge X \in \mathrm{instAS}((\!| SX |\!) \restriction^{\alpha\mathtt{st}} t) \\ \bullet\ T_F(t, X) \end{array} \right\} \end{array} \right\}$$

[Lemma 8]

$$= \bigcup \left\{ \begin{array}{l} \mathtt{st}, \mathtt{c}, SX, t, X \mid \\ \quad \left(\begin{array}{l} (\mathtt{st}, \mathtt{c}) \in cstraces^{\mathsf{a}}(\mathtt{SP}) \land \\ SX \in csacceptances^{\mathsf{a}}(\mathtt{SP}, (\mathtt{st}, \mathtt{c})) \land \\ t \in \mathrm{instT}(\mathtt{st}, \mathtt{c}) \land X \in \mathrm{instAS}(\llparenthesis SX \rrparenthesis \upharpoonright^{\alpha \mathtt{st}} t) \end{array} \right) \\ \bullet\ T_F(t, X) \end{array} \right\}$$

[property of sets]

$$= \bigcup \{ t, X \mid t \in traces(\mathtt{SP}) \land X \in acceptances(\mathtt{SP}, t) \bullet T_F(t, X) \}$$

[Proposition 4]

$$= ExhaustA_F(\mathtt{SP}) \qquad [\text{definition of } ExhaustA_F]$$

□

Finally, we address tests for processes; Theorem 4 is similar to Theorem 2.

Theorem 4 Symbolic exhaustivity - processes - *conf* *For every alphabet* a*,*

$$Exhaust_F(\mathtt{SP}) = \bigcup \{ \mathtt{ST} : \mathtt{SExhaust}^{\mathsf{a}}_{\mathsf{F}}(\mathtt{SP}) \bullet \mathrm{instP}(\mathtt{ST}) \}$$

Proof Similar to the proof of Theorem 2, but it uses Lemma 11. □

In [16], we have proved that process refinement in *Circus* can be characterised by the conjunction of traces refinement and *conf*. Therefore, Theorems 2 and 4 allow us to conclude that the union of the exhaustive sets for traces refinement and for *conf*, as presented in Definitions 14 and 18, is an exhaustive test set for process refinement in *Circus*. In addition, we can then conclude that, if the testability hypotheses hold, then the SUT passes this set of tests if, and only if, the *Circus* model used to define the tests is refined by the *Circus* model of the SUT. Precisely, we say that a SUT passes a test set TS, if it passes every test t in TS. In addition, a SUT passes a test t if the verdict of n executions of t, as characterised in Definition 10, is not a failure. The number n comes from the complete testing assumption.

Together these results justify the use of the symbolic tests as a starting point for selection strategies. The symbolic tests benefit from a concise and comprehensive account of the data operations and their properties. Coverage criteria can consider both the concrete labelled transition system and the symbolic transition system. With the latter, we have a handle for coverage of the data operations and the structure of the constraints. Actually, the constraints provide a basis for the definition of the so-called equivalence classes, or uniformity subdomains, used in the testing literature.

6 Related work on formal testing

As far as we know, there is no formal theory of symbolic testing for a model-based language that covers predicative specification of data types and process algebraic specification of behaviour. Our results lay the foundation to justify the soundness of elaborate testing techniques based on this important class of languages. They put forward a clear path for the use of coverage criteria that are in tune with the nature of such comprehensive notations.

On the other hand, several works indicate routes to apply the theory presented here. We discuss some of them in the sequel.

Z Most testing methods for Z refer to the seminal work for VDM by Dick and Faivre [21] that is applicable to all model-based notations. The formulae of VDM specifications are relations on states described by operations expressed in first-order predicate calculus. In [21], these relations are reduced to a disjunctive normal form (DNF), creating a set of disjoint subrelations. Each of them yields a set of constraints that describe a single disjoint test domain. As VDM is state-based, a finite state automaton is extracted using the results of the partition analysis of the operations to partition the states. From this automaton, test sequences are produced to ensure coverage of the paths. The notion of test sequence is strongly related to the state orientation of the specification and the fact that states are hidden: testing an operation for a given subrelation requires reaching an adequate state via some sequence of inputs, and to observe the resulting state by using other operations that provoke outputs [35]. This method was transposed to Z by Helke et al. [32], with the DNF partition of the schemas and the elimination of the unsatisfiable cases automated using Isabelle.

Major problems with this approach are explosion of the number of test cases induced by the DNF reduction and proliferation of unfeasible cases. The first problem can be overcome by limiting the reduction, similarly to what was done in [46] for axiom unfolding, or by applying specific decomposition strategies such as the one described in [35]. The second problem can be alleviated by powerful constraint solvers or theorem provers [32,8], but this remains to be investigated in the context of Z. In the context of *Circus*, we plan to explore this issue, since the Z operations are captured in the constraints of the symbolic traces and acceptances presented previously.

In [12] Carrington and Stock presented a framework for Z-based testing with a choice of strategies, including reduction to DNF. The key element was the construction of abstract descriptions of test data called test templates: Z descriptions of test cases. The definition of templates for each operation was based on the Valid Input Space template, which roughly corresponds to the precondition. Further templates were derived according to the chosen testing strategy. Derivation of test cases was not automatic; potential problems with unsatisfiable templates were not mentioned. More recently, a tool that partially automates this approach has been developed; it relies on Z/Eves (a Z theorem prover), ZLive (a Z animator), and customised algorithms [20].

The application of test templates and automatic partition of schemas for systems of industrial size have been problematic due to the lack of efficient constraint solvers and theorem provers. Given the progress recently achieved for these tools, these techniques are worth revisiting for integration in a global testing method for *Circus* guided by our symbolic tests.

CSP There has been a lot of interest in testing based on CSP. As already said, however, it was only recently that we have ourselves defined a testing theory for CSP based on its notion of failures(-divergences) refinement.

Peleska and Siegel [55,56] applied CSP-based testing, but did not address the gap between the system under test and the CSP model; their test sets are inspired by, but not in direct correspondence with, their theoretical definitions. Schneider [59] defined conformance relations for CSP based on testing and showed how model checking can be used to establish such relations. The work of Srivatanakul et al. [63] applies mutation-testing techniques to a CSP model, and uses the mutants that satisfy the properties of interest as a basis for clarification of requirements. More recently, CSP was used by Nogueira et al. to formalise a notion of conformance traditionally associated with input-output labelled transition systems [50], with refinement model checking used for generation and selection of tests.

Our theory for CSP could be used in the context of *Circus*, but does not provide a clear route to the treatment of the model-based data type definitions. As explained before, symbolic tests bring the possibility to take into account coverage criteria of both behavioural and data type descriptions. This is the main contribution of the work presented here.

Conversely, our symbolic account of traces, initials, and acceptances can be used to model CSP processes. The operational semantics of a CSP process, however, leads to trivial symbolic specifications, as there are no abstract data operations intermingled in a CSP process, and all values are fully evaluated.

State machines A coverage criterion that has been widely used for Finite State Machines (FSM) is transition coverage [42]. It is justified by the assumption that the SUT behaves like some (unknown) FSM: there is no memory associated to states and, from a given state, the transitions associated with an input have the same effect, whatever the execution history.

Extended FSM (EFSM) have been used for describing systems with an infinite or a very large number of states (SUT). There are numerous variants. Similar extensions have been proposed for Labelled Transition Systems (LTS) under the name of Symbolic Labelled Transition Systems (SLTS). For EFSMs and SLTSs, the above justification of transition coverage does not hold, since the behaviour of a transition depends on the values of the variables in the current state, and those depend on the previous transitions. In addition, guards and symbolic events lead to reachability and feasibility problems [37]. Some approaches have been proposed for generating test cases from specifications in full LOTOS and SDL [70,60]. The idea was to map the specification into an EFSM, and apply classical program testing strategies for selection.

Later on, taking advantage of testing methods for algebraic data types, Gaudel and James suggested an integrated approach for full LOTOS, where behavioural descriptions and algebraic data types are considered [31]. The approach produces interesting boundary test cases, and has been generalised in [43] to Input-Output Transition Systems with parameters and LOTOS-like data types. In that work, the idea of propagating constraints on communication variables and parameters along traces to get symbolic tests is introduced. A prototype test-generation tool was developed, but failed to give convincing results due to the lack of a powerful constraint solver, and the problem of infeasible traces. Infeasibility has been addressed later for SDL in [37].

The ideas in these works have influenced our definition of the *Circus* exhaustive test set. The selection criteria and test generation algorithms, however, cannot be directly transposed to *Circus*. Its flexible integration of Z and CSP to cater for programming architectures and constructs, and its unstructured definition of data operations based on predicates, rather than axioms, impose a new challenge. In addition, the semantic framework of *Circus* and its notion of conformance, namely, refinement, are entirely different.

A theory of testing based on CO-OPN specifications was stated by Peraire et al. in [57]. CO-OPN uses algebraic structures and object-based concepts to define data types and Petri nets to handle concurrency. The implementation relation is observational equivalence, and tests are Hennessy-Milner formulas labelled by true or false. A notion of exhaustive test set associated to a specification has been stated in [57]. Due to the different nature of the formalism and conformance relation, the tests are necessarily very different from those presented here. The operational semantics of CO-OPN has been implemented in Prolog, providing a powerful prototyping tool: the development approach there is based on prototyping and testing, and is thus different from the one considered in *Circus*, which includes refinements and coexistence of abstract specifications and executable programs.

In a more recent work, Lucio et al. [44] present a test selection language for CO-OPN specifications, which is used to capture the test intention of the test engineer: the test selection language combines Hennessy-Milner Logic and constraints. It serves as basis for a test-generation tool. The use of this tool is reported in [9], which considers model-based testing for BPMN (Business Process Modelling Notation). The approach to define and use test intentions is potentially relevant in the context of *Circus*.

A symbolic version of the *ioco* relation [69] is used by Frantzen et al. [25] to handle Input-Output Symbolic Transition Systems (IOSTS), where states are locations decorated with variables, and transitions are labelled by guards, by events with interaction variables, and by assignment actions. The main result is a conformance relation between IOSTS and input-enabled IOSTS. The SUT is assumed to accept every input in every state. The models and relations are different from ours, but there are similar notions of symbolic extended traces, with a formula constraining the interaction variables, and another constraint on the update of the state variables. They propose location coverage: a sort of symbolic state coverage. We can invent a similar coverage for *Circus*, but given its analogy with statement coverage in program testing, it is not clear it would provide sufficient fault-detection power. It is not the objective of [25] to explore the definition of symbolic tests or exhaustive symbolic test sets as a basis for test generation.

UML statecharts Due to their similarity to EFSMs and SLTSs, UML statecharts are very popular as bases for testing. Of relevance to *Circus* is that by Hierons et al. in [34], which considers a hybrid specification language where a system is described using statecharts, whose states, guards, and operations are specified in Z. In this work, Z state decomposition techniques for testing are used to transform the associated statechart (EFSM), which is then used to generate test cases. The focus of the work is to circumvent false negatives arising from coincidental correctness; for that, notions of distinguishable states and transitions are introduced, and used to generate tests.

In [4,3], a testing method for statecharts that takes into account hierarchy and concurrency is proposed by Bodganov and Holcombe. It provides selection criteria that significantly reduce the size of test sets in an effective way. The work by Briand et al. in [6] considers flattened deterministic statecharts where states are objects or object clusters, and transitions are labelled by guards and actions defined by pre and postconditions expressed in OCL. Normalisation is used to handle the constraints and guide the test-case generation; the construction of constraints is described by an algorithm.

All these results related to UML statecharts are of high practical relevance, but their theoretical background has not been developed yet. The testing theory we developed here for *Circus* can be adapted to be used as a basis to justify the pragmatic techniques proposed.

In [47] Maassink et al. study a version of statecharts named UMLSC. A testing theory, in the sense of [49], that is, for models rather than systems, as well as a notion of conformance and a test-case generation algorithm, are presented. The theoretical framework developed follows the same pattern as ours, but the considered language and conformance relation are very different. In addition, symbolic treatment of data is not covered, and the consideration of data values and variables is left as future work.

7 Conclusions

As an experimental technique, testing is a pragmatic activity. The justification of the testing process, however, can be formalised, as already widely discussed in the literature. What we provide here is the theoretical foundation to achieve such formalisation for an important class of refinement languages for state-rich reactive systems. This is in contrast with approaches that focus on tool development and experiments, but do not justify their conclusions.

In concrete terms, our main result is a foundation for model-based testing techniques for a rich and flexible modelling language that can be used for data and behavioural modelling, namely, *Circus*. Our testability hypotheses are standard: we assume that the SUT behaves like some unknown *Circus* process, and adopt the complete testing assumption to handle nondeterminism. Divergence is treated as deadlock, since this is the only and standard way in which divergence is perceived or interpreted in testing experiments.

The basis of our work is an operational semantics that gives a symbolic account of the evolution of a system; it captures internal state changes, and associated constraints on interactions. Nondeterminism in interaction patterns is captured in the standard way, but nondeterminism in data values is handled separately, and reconciled in the semantics of parallelism.

Using the operational semantics, we have provided symbolic characterisations of traces, initials, and acceptances. The challenge was to determine unique (up to predicate equivalence) symbolic representatives of these sets. Our solution is to fix an ordered alphabet of symbolic variables. That allows a uniform account of these sets, and simplifies the management of names by avoiding renaming complications. It also simplifies the symbolic characterisation of the forbidden continuations (that is, the complement of the set of initials). Our definitions of symbolic traces, initials, and acceptances are in direct correspondence with the standard characterisations of these notions in a concrete setting (like that of CSP, for instance).

The sets of symbolic traces, initials and acceptances have been used to define symbolic tests and exhaustive test sets for traces refinement and *conf*. Detailed algebraic proofs of exhaustiveness have been presented; all theorems and lemmas are proved in detail and this provides strong validation of the definitions. Together the symbolic exhaustive test sets ensure exhaustivity with respect to process refinement in *Circus*, given the testability hypotheses. The symbolic tests and exhaustive test sets are ideal bases for work on optimisation, selection criteria, and test-generation algorithms that take into account both data operations and interaction patterns.

The notions of instantiation, especially that for the symbolic acceptance sets, are rather subtle; we have, therefore, provided a detailed formalisation. Instantiation is at the centre of our proof of exhaustivity for the symbolic test sets, but is also of practical interest. The instantiation functions will be essential components of test-generation algorithms.

The exhaustive test sets are infinite for any non-trivial specification. We plan to investigate various approaches to address the problem of selection of finite test sets, and the automatic generation of test cases. Since we have a symbolic version of the tests, with labels constraining communicated values, it is natural to consider strategies based on constraints decomposition and solving. Our theory, however, puts us in a strong position to go further and address the challenging problem of providing coverage of complex internal data operations, and justify the soundness of the techniques.

There are two fundamental properties that the selected test sets need to satisfy: validity and unbias. Validity ensures that if the selected tests pass, then those in the exhaustive set do as well. On the other hand, a set of selected tests is unbiased if whenever the tests in the exhaustive test set pass, so do its own tests. In summary, we have to prove that, under the given selection hypothesis, the selected test set rejects all incorrect SUT and does not reject any correct SUT. For some of the selection criteria that we will investigate, validity and unbias is obvious, but for others proofs are expected to be tricky. Some authors use a different terminology: unbias is sometimes called soundness, and validity is sometimes called exhaustivity [69].

Because we have identified testability hypotheses, we will proceed along the same lines with the idea of formally expressing testing strategies and criteria as hypotheses on the SUT. This was first proposed by Bernot et al. [2], and then recommended by Hierons in [36], since test hypotheses make it possible to formalise and compare test sets and test criteria.

We perceive two approaches for selection. The first defines subsets of the symbolic exhaustive test set, and the second is guided by the text of the *Circus* specification. Our long term goal is the identification of a plethora of sound selection strategies and corresponding test-case generation algorithms for *Circus*, with a corresponding evaluation of their effectiveness. This work will lead to an evaluation of our testing theory in terms of the difficulty to use it to prove soundness of the techniques.

A Operational semantics: choice and parallelism

For an internal choice $\mathtt{A}_1 \sqcap \mathtt{A}_2$, silent transitions are available to either $\mathtt{A}_1$ or $\mathtt{A}_2$ (in a configuration with the same constraint and state assignment).

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1 \sqcap \mathtt{A}_2) \xrightarrow{\epsilon} (\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1)} \qquad \frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1 \sqcap \mathtt{A}_2) \xrightarrow{\epsilon} (\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_2)} \tag{12}$$

For an external choice, an extra syntactical construct is used to keep track of local state changes that do not decide the choice, and may need to be discarded when that happens. As an example, we consider the external choice $x := x + 1;\ d_1!x \to \mathbf{Skip}\ \Box\ x := x + 2;\ d_2!x \to \mathbf{Skip}$. The choice is only decided when a communication via either d_1 or d_2 occurs. Before reaching that point, however, each of the actions $x := x + 1;\ d_1!x \to \mathbf{Skip}$ and $x := x + 2;\ d_2!x \to \mathbf{Skip}$ can evolve silently by executing their assignments. In spite of that, the effect of the assignment $x := x + 1$, for example, is made permanent only when, and if, a communication on d_1 decides the choice. Similarly, if the choice is determined by a communication on d_2, then $x := x + 2$ becomes permanent (and the effect of $x := x + 1$ is ignored).

The first step in the execution of an external choice, therefore, is a silent transition that introduces a loc clause in each action in the choice to record the current constraint and state assignment locally. This is captured by the transition rule (13) below. The choice operator $\boxplus$, which is defined for the operational semantics, is introduced to indicate that a local state is already recorded, but a choice has not yet been made.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1 \Box \mathtt{A}_2) \xrightarrow{\epsilon} (\mathtt{c} \mid \mathtt{s} \models (\mathtt{loc}\ \mathtt{c} \mid \mathtt{s} \bullet \mathtt{A}_1) \boxplus (\mathtt{loc}\ \mathtt{c} \mid \mathtt{s} \bullet \mathtt{A}_2))} \tag{13}$$

The transition rule (14) for external choice determines that, if the first action in the choice terminates, then it is possible to decide the choice in favour of that action: its local constraint and state become those of the target configuration. We omit the similar rule that covers the case when the second action in the choice terminates. (External choice is commutative.)

$$\frac{c_1}{(\mathtt{c} \mid \mathtt{s} \models (\mathtt{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{Skip}) \boxplus (\mathtt{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A})) \xrightarrow{\epsilon} (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{Skip})} \tag{14}$$

The transition rule (15) considers the possibility of silent evolution of the first action $\mathtt{A}_1$ in the choice. In this case, the evolution is recorded locally in the choice. Again, we omit the similar rule for silent evolution of $\mathtt{A}_2$.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)}{\left(\begin{array}{l}\mathtt{c} \mid \mathtt{s}\\ \models\\ \quad\left(\begin{array}{l}(\mathtt{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1)\\ \boxplus\\ (\mathtt{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)\end{array}\right)\end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l}\mathtt{c} \mid \mathtt{s}\\ \models\\ \quad\left(\begin{array}{l}(\mathtt{loc}\ \mathtt{c}_3 \mid \mathtt{s}_3 \bullet \mathtt{A}_3)\\ \boxplus\\ (\mathtt{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)\end{array}\right)\end{array}\right)} \tag{15}$$

Finally, we have rule (16) for the case in which a labelled transition (that is, a transition whose label is different from ϵ) is possible for $\mathtt{A}_1$ in its current local constraint and state. In this case, the choice for $\mathtt{A}_1$ eliminates the $\boxplus$ operator; the target configuration is that of the transition from $\mathtt{A}_1$.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad l \neq \epsilon}{(\mathtt{c} \mid \mathtt{s} \models (\mathtt{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \boxplus (\mathtt{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)} \tag{16}$$

Example 17 We consider below the action *CashC*, in a configuration characterised by the constraint $\mathtt{w}_0 \in \mathtt{T}$, where T is the type of *noteBank*, and the state assignment $\mathtt{noteBank} := \mathtt{w}_0$. We omit ϵ labels in examples.

$$(\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \models \mathtt{CashC})$$

$$\longrightarrow \left(\begin{array}{l}\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0\\ \models\\ \quad\left(\begin{array}{l}(\mathtt{loc}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \bullet \mathtt{refill} \to \ldots)\\ \boxplus\\ (\mathtt{loc}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \bullet \mathtt{disp?a} \to \ldots)\end{array}\right)\end{array}\right) \text{[Rule (13)]}$$

Since there are no internal actions possible in any of the choices, there are only two possible transitions from the above configuration. Either we have a synchronisation on *refill* or on *disp*. The transition for the synchronisation on *refill* is below; there is no communicated value, so no extra symbolic variable.

$$\xrightarrow{\mathtt{refill}} \left(\begin{array}{l}\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0\\ \models\\ \quad \mathtt{noteBank} := \{\, 10 \mapsto \mathtt{cap}, 20 \mapsto \mathtt{cap}, 50 \mapsto \mathtt{cap} \,\}\ ;\ \mathtt{CashC}\end{array}\right)$$

[Rules (16) and (5)]

We do not present here the rule for sequence, but it is standard, and it allows the progress of the first action until it finishes (reduces to a configuration with action Skip), when then the second action takes over. In our example, our first action is an assignment, whose transition rule (3) was discussed above.

$$\longrightarrow \begin{pmatrix} \mathtt{w_0} \in \mathtt{T} \land \mathtt{w_1} = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \mid \mathtt{noteBank} := \mathtt{w_1} \\ \models \\ \quad \mathtt{Skip} \,;\; \mathtt{CashC} \end{pmatrix} \qquad \text{[Rule (3)]}$$

$$\longrightarrow \begin{pmatrix} \mathtt{w_0} \in \mathtt{T} \land \mathtt{w_1} = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \mid \mathtt{noteBank} := \mathtt{w_1} \\ \models \\ \quad \mathtt{CashC} \end{pmatrix}$$

...

We consider the transition labelled by a communication on disp below.

$$\xrightarrow{\mathtt{disp?w_1}} (\mathtt{w_0} \in \mathtt{T} \land \mathtt{w_1} \in \mathbb{N}_1 \mid \mathtt{noteBank} := \mathtt{w_0} \models \mathbf{var}\ \mathtt{notes} : \mathtt{Cash} \bullet \ldots)$$
$$\text{[Rules (16) and (6)]}$$

The omitted rule for variable blocks [72], determines the next step. □

Local information is also needed to define the semantics of parallelism. Each parallel action needs a local record of the current value of the state and, additionally, the set of the names of the variables to which it has write access. This is because, as already explained, each parallel action uses independently the state at the start of the parallelism. Also, at the end of the parallelism, it is their effect on the variables in their associated sets that defines the state.

So, the first (silent) step of a parallelism $\mathtt{A_1} \llbracket \mathtt{x_1} \mid \mathtt{cs} \mid \mathtt{x_2} \rrbracket \mathtt{A_2}$ introduces a new parallel operator that records only the synchronisation set cs, and par clauses in the parallel actions to record the local information. This is captured in the transition rule (17) shown below. The sets $\mathtt{x_1}$ and $\mathtt{x_2}$ are assumed to partition all the state components in scope; this is not necessarily the case in a *Circus* model, but can be easily enforced by pre-processing.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A_1} \llbracket \mathtt{x_1} \mid \mathtt{cs} \mid \mathtt{x_2} \rrbracket \mathtt{A_2}) \xrightarrow{\epsilon} \begin{pmatrix} \mathtt{c} \mid \mathtt{s} \\ \models \\ \begin{pmatrix} (\mathtt{par}\ \mathtt{c} \mid \mathtt{s} \mid \mathtt{x_1} \bullet \mathtt{A_1}) \\ \llbracket \mathtt{cs} \rrbracket \\ (\mathtt{par}\ \mathtt{c} \mid \mathtt{s} \mid \mathtt{x_2} \bullet \mathtt{A_2}) \end{pmatrix} \end{pmatrix}} \qquad (17)$$
$$out\alpha s = (x_1', x_2')$$

The rule (18) below for the special parallel operator states that, when both parallel actions terminate (that is, are reduced to Skip), then the whole parallelism terminates. The constraint in the target configuration is the conjunction of the local constraints $\mathtt{c_1}$ and $\mathtt{c_2}$ of the parallel actions. The fact that they are jointly satisfiable is a hypothesis of the rule. The state assignment is also the conjunction of the local states $\mathtt{s_1}$ and $\mathtt{s_2}$, but in this case, first it is necessary to quantify away in $\mathtt{s_1}$ the variables $\mathtt{x_2'}$ whose values are to be determined by $\mathtt{s}_2$. Symmetrically, $\mathtt{x}'_1$ is quantified away from $\mathtt{s}_2$.

$$\frac{c_1 \wedge c_2}{\left(\begin{array}{l} \mathtt{c}_1 \wedge \mathtt{c}_2 \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{c}_1 \mid \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{Skip}) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{c}_2 \mid \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} \mathtt{c}_1 \wedge \mathtt{c}_2 \mid (\exists\, \mathtt{x}'_2 \bullet \mathtt{s}_1) \wedge (\exists\, \mathtt{x}'_1 \bullet \mathtt{s}_2) \\ \models \\ \quad \mathbf{Skip} \end{array}\right)} \quad (18)$$

We observe that this rule applies only to configurations in which the constraint of the parallelism is the conjunction of the local constraints $\mathtt{c}_1$ and $\mathtt{c}_2$ of the parallel actions. While the local state assignments are only relevant for their associated parallel actions, the local constraints are relevant for the parallelism as it evolves. This is because some of the symbolic variables are used in communications of the parallel action, and so the restriction on them needs to be recorded. The next two rules further clarify this point.

When there is a labelled transition for the first parallel action $\mathtt{A}_1$ with a label $\mathtt{l}$ that is either ϵ or a communication over a channel chan $\mathtt{l}$ that is not in the synchronisation set, the rule (19) for the extra parallel operator indicates that the whole parallelism evolves accordingly. (The second parallel action does not need to participate in, and is not affected by, such a transition.) The local state information for $\mathtt{A}_1$ is updated in the parallelism. We omit the symmetric rule for the second parallel action.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad l = \epsilon \vee \mathrm{chan}\, l \notin cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_2}{\left(\begin{array}{l} \mathtt{c}_1 \wedge \mathtt{c}_2 \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{c}_1 \mid \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{c}_2 \mid \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\mathtt{l}} \left(\begin{array}{l} \mathtt{c}_3 \wedge \mathtt{c}_2 \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{c}_3 \mid \mathtt{s}_3 \mid \mathtt{x}_1 \bullet \mathtt{A}_3) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{c}_2 \mid \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right)} \quad (19)$$

The transition rule (20), the last that we present for the special parallel operator, considers the case in which there is a transition for the first parallel action with a label $\mathtt{d?w}_1$ and a transition for the second action with a corresponding label $\mathtt{d!w}_2$, and the channel d is in the synchronisation set. In this case, progress requires that the parallel actions agree on the communication and synchronise. The output determines the value w_2 to be communicated, therefore the parallel action evolves with the event $\mathtt{d!w}_2$ determined by the output. Accordingly, the constraint $c_3 \wedge c_4 \wedge w_1 = w_2$ requires that equating w_1 to w_2, in the context of the existing constraints c_3 on w_1 and c_4 on w_2, is satisfiable. This becomes the constraint of the target configuration. The equality $\mathtt{w}_1 = \mathtt{w}_2$ is also kept in the local constraints, so that the constraint of the parallelism remains the conjunction of the local constraints.

$$\frac{\begin{array}{c}(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d?w}_1} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{d!w}_2} (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \\ d \in cs \quad c_1 \wedge c_2 \quad c_3 \wedge c_4 \wedge w_1 = w_2\end{array}}{\left(\begin{array}{l}\mathtt{c}_1 \wedge \mathtt{c}_2 \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c}(\mathtt{par}\ \mathtt{c}_1 \mid \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{c}_2 \mid \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2)\end{array}\right)\end{array}\right) \xrightarrow{\mathtt{d!w}_2} \left(\begin{array}{l}\mathtt{c}_3 \wedge \mathtt{c}_4 \wedge \mathtt{w}_1 = \mathtt{w}_2 \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c}(\mathtt{par}\ \mathtt{c}_3 \wedge \mathtt{w}_1 = \mathtt{w}_2 \mid \mathtt{s}_3 \mid \mathtt{x}_1 \bullet \mathtt{A}_3) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{c}_4 \wedge \mathtt{w}_1 = \mathtt{w}_2 \mid \mathtt{s}_4 \mid \mathtt{x}_2 \bullet \mathtt{A}_4)\end{array}\right)\end{array}\right)} \tag{20}$$

The label $\mathtt{d!w}_2$ of the parallelism records an output because, since one of the parallel actions produces an output, synchronisation constrains the communication of the other action to the output value, and so the whole parallelism produces that output. The same comments apply when we have communications $\mathtt{d!w}_1$ and $\mathtt{d?w}_2$, or two outputs $\mathtt{d!w}_1$ and $\mathtt{d!w}_2$, which need to agree on a single value. The rule for when the parallel actions have transitions with labels $\mathtt{d?w}_1$ and $\mathtt{d?w}_2$ is similar, but in this case the transition of the parallelism has label $\mathtt{d?w}_2$. Since both parallel actions accept inputs, the whole parallelism accepts the inputs that are acceptable by both of them.

Example 18 For the parallelism in the main action of *CashMachine*, again in the initial configuration for this process, we have the possible sequence of transitions sketched below, where we first have a(n independent) communication on *refill* from *CashC*, and then a synchronisation on *disp*.

$$\left(\begin{array}{l}\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \mathtt{CardV}\ [\![\ \{\,\} \mid \{\!|\ \mathtt{disp}, \mathtt{ok}\ |\!\} \mid \{\mathtt{noteBank}\}\ ]\!]\ \mathtt{CashC}\end{array}\right)$$

$$\longrightarrow \left(\begin{array}{l}\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \left(\begin{array}{l}(\mathtt{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\,\} \bullet \mathtt{CardV}) \\ \qquad [\![\ \{\!|\ \mathtt{disp}, \mathtt{ok}\ |\!\}\ ]\!] \\ (\mathtt{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\mathtt{noteBank}\} \bullet \mathtt{CashC})\end{array}\right)\end{array}\right)$$

[Rule (17)]

$$\longrightarrow \left(\begin{array}{l}\mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \left(\begin{array}{l}(\mathtt{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\,\} \bullet \mathtt{CardV}) \\ \qquad [\![\ \{\!|\ \mathtt{disp}, \mathtt{ok}\ |\!\}\ ]\!] \\ \left(\begin{array}{l}\mathtt{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\mathtt{noteBank}\} \bullet \\ \quad \left(\begin{array}{l}(\mathtt{loc}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \bullet \mathtt{refill} \to \ldots) \\ \boxplus \\ (\mathtt{loc}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \bullet \mathtt{disp?a} \to \ldots)\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

[Rule (13)]

$$\xrightarrow{\mathtt{refill}} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\,\} \bullet \mathtt{CardV}) \\ \qquad [\![\{\!|\, \mathtt{disp}, \mathtt{ok} \,|\!\}]\!] \\ \left(\begin{array}{l} \mathsf{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\mathtt{noteBank}\} \bullet \\ \quad \mathtt{noteBank} := \{\, 10 \mapsto \mathtt{cap}, \ldots \}\ ;\ \mathtt{CashC} \end{array}\right) \end{array}\right) \end{array}\right)$$

[Rules (19), (16), and (5)]

$$\longrightarrow \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \mathtt{w}_1 = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\,\} \bullet \mathtt{CardV}) \\ \qquad [\![\{\!|\, \mathtt{disp}, \mathtt{ok} \,|\!\}]\!] \\ \left(\begin{array}{l} \mathsf{par} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \mathtt{w}_1 = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \\ \mid \mathtt{noteBank} := \mathtt{w}_1 \\ \mid \{\mathtt{noteBank}\} \end{array}\right) \bullet \\ \quad \mathtt{Skip}\ ;\ \mathtt{CashC} \end{array}\right) \end{array}\right) \end{array}\right)$$

[Rules (19) and (3)]

$$\longrightarrow \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \mathtt{w}_1 = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \mid \mathtt{noteBank} := \mathtt{w}_0 \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{par}\ \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{noteBank} := \mathtt{w}_0 \mid \{\,\} \bullet \mathtt{CardV}) \\ \qquad [\![\{\!|\, \mathtt{disp}, \mathtt{ok} \,|\!\}]\!] \\ \left(\mathsf{par} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \mathtt{w}_1 = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \\ \mid \mathtt{noteBank} := \mathtt{w}_1 \\ \mid \{\mathtt{noteBank}\} \end{array}\right) \bullet \mathtt{CashC} \right) \end{array}\right) \end{array}\right)$$

...

$$\xrightarrow{\mathtt{incard}?\mathtt{w}_8!\mathtt{w}_9?\mathtt{w}_{10}}$$

...

$$\longrightarrow \left(\begin{array}{l} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \ldots \wedge \mathtt{w}_{10} \in \mathbb{N}_1 \\ \mid \mathtt{noteBank} := \mathtt{w}_0 \end{array}\right) \\ \models \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} \mathsf{par} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \ldots \wedge \mathtt{w}_{10} \in \mathbb{N}_1 \\ \mid \mathtt{noteBank} := \mathtt{w}_0;\ \mathbf{var}\, \mathsf{c}, \mathsf{a} := \mathtt{w}_8, \mathtt{w}_{10} \\ \mid \{\,\} \end{array}\right) \\ \quad \bullet\ \mathtt{disp}!\mathsf{a} \to \mathsf{ok} \to \mathsf{outcard}!\mathsf{c} \to \mathtt{CardV} \end{array}\right) \\ \qquad [\![\{\!|\, \mathtt{disp}, \mathtt{ok} \,|\!\}]\!] \\ \left(\mathsf{par} \left(\begin{array}{l} \mathtt{w}_0 \in \mathtt{T} \wedge \mathtt{w}_1 = \{\, 10 \mapsto \mathtt{cap}, \ldots \} \\ \mid \mathtt{noteBank} := \mathtt{w}_1 \\ \mid \{\mathtt{noteBank}\} \end{array}\right) \bullet \mathtt{CashC} \right) \end{array}\right) \end{array}\right)$$

$$\xrightarrow{\mathtt{disp!w_{13}}} \left(\begin{array}{l} \left(\begin{array}{l} \mathtt{w_0} \in \mathtt{T} \wedge \ldots \wedge \mathtt{w_{13}} = \mathtt{w_{10}} \wedge \mathtt{w_{13}} = \mathtt{w_{14}} \\ \mid \mathtt{noteBank} := \mathtt{w_0} \end{array} \right) \\ \models \\ \left(\begin{array}{l} \left(\begin{array}{l} \mathtt{par} \left(\begin{array}{l} \mathtt{w_0} \in \mathtt{T} \wedge \ldots \wedge \mathtt{w_{13}} = \mathtt{w_{10}} \wedge \mathtt{w_{13}} = \mathtt{w_{14}} \\ \mid \mathtt{noteBank} := \mathtt{w_0};\ \mathbf{var}\ \mathtt{c}, \mathtt{a} := \mathtt{w_8}, \mathtt{w_{10}} \\ \mid \{\} \end{array} \right) \\ \quad \bullet\ \mathtt{ok} \rightarrow \mathtt{outcard!c} \rightarrow \mathtt{CardV} \end{array} \right) \\ \quad [\![\{\!| \mathtt{disp}, \mathtt{ok} |\!\}]\!] \\ \left(\begin{array}{l} \mathtt{par} \left(\begin{array}{l} \mathtt{w_0} \in \mathtt{T} \wedge \ldots \wedge \mathtt{w_{13}} = \mathtt{w_{14}} \\ \mid \mathtt{noteBank} := \mathtt{w_1}\ ;\ \mathbf{var}\ \mathtt{a} := \mathtt{w_{14}} \\ \mid \{\mathtt{noteBank}\} \end{array} \right) \\ \quad \bullet\ (\mathbf{var}\ \mathtt{notes} : \mathtt{Cash} \bullet \ldots) \end{array} \right) \end{array} \right) \end{array} \right)$$

[Rules (20), (6), (16), and (5)]

...

Since the parallelism does not finish, the par operator is never eliminated. As said before, synchronisation communications, like *refill*, do not require the use of extra symbolic variables. Also, we observe that in a communication like *incard*?*c*.(*pin c*)?*a*, we do not explicitly indicate (*pin c*) as an output, because intuitively this is a synchronisation on a specific value defined by the function *pin*. Semantically, however, this is equivalent to an output and is recorded as such in the label of the transition that models the communication. □

The complete set of transition rules can be found in [72].

Acknowledgements We are grateful to the Royal Society of London, who supported our collaboration through funding for an International Joint Project.

References

1. Abrial, J.R.: B#: toward a synthesis between Z and B. In: D. Bert, J.P. Bowen, S. King, M. Waldén (eds.) ZB, *Lecture Notes in Computer Science*, vol. 3582, pp. 168 – 177. Springer-Verlag (1996)
2. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications: a theory and a tool. Software Engineering Journal **6**(6), 387 – 405 (1991)
3. Bogdanov, K., Holcombe, M.: Refinement in statechart testing: Research articles. Software Testing, Verification and Reliability **14**(3), 189–211 (2004)
4. Bogdanov, K., Holcombe, M., Singh, H.: Automated test set generation for statecharts. In: FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method, pp. 107–121. Springer-Verlag (1999)
5. Bougé, L., Choquet, N., Fribourg, L., Gaudel, M.C.: Test set generation from algebraic specifications using logic programming. Journal of Systems and Software **6**(4), 343 – 360 (1986)
6. Briand, L.C., Labiche, Y., Cui, J.: Automated support for deriving test requirements from UML statecharts. Journal of Software and Systems Modeling pp. 399 – 423 (2005)
7. Brinksma, E.: A theory for the derivation of tests. In: Protocol Specification, testing and Verification VIII, pp. 63 – 74. North-Holland (1988)

8. Brucker, A.D., Rittinger, F., Wolff, B.: Hol-z 2.0: A proof environment for z-specifications. Journal of Universal Computer Science **9**(2), 152–172 (2003)
9. Buchs, D., Lucio, L., Chen, A.: Model checking techniques for test generation from business process models. In: 14th Ada-Europe International Conference on Reliable Software Technologies, Lecture Notes in Computer Science, pp. 59 – 74. Springer-Verlag (2009)
10. Butler, M.J.: csp2B: A Practical Approach to Combining CSP and B. Formal Aspects of Computing **12**(3), 182 – 198 (2000)
11. Butterfield, A., Sherif, A., Woodcock, J.C.P.: Slotted Circus: A UTP-family of reactive theories. In: International Conference on Formal Enginneering, *Lecture Notes in Computer Science*, vol. 4591, pp. 75 – 97. Springer-Verlag (2007)
12. Carrington, D., Stocks, P.: A tale of two paradigms: Formal methods and software testing. In: J.P. Bowen, J.A. Hall (eds.) Z User Workshop, Workshops in Computing, pp. 51 – 68. Springer-Verlag (1994)
13. Cavalcanti, A.L.C., Clayton, P., O'Halloran, C.: Control Law Diagrams in *Circus*. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, *Lecture Notes in Computer Science*, vol. 3582, pp. 253 – 268. Springer-Verlag (2005)
14. Cavalcanti, A.L.C., Gaudel, M.C.: Testing for Refinement in CSP. In: 9th International Conference on Formal Engineering Methods, *Lecture Notes in Computer Science*, vol. 4789, pp. 151 – 170. Springer-Verlag (2007)
15. Cavalcanti, A.L.C., Gaudel, M.C.: Testing for Refinement in *Circus* – Extended version. Tech. rep., University of York (2009). `www-users.cs.york.ac.uk/~alcc/CG09.pdf`
16. Cavalcanti, A.L.C., Gaudel, M.C.: A note on traces refinement and the *conf* relation in the Unifying Theories of Programming. In: A. Butterfield (ed.) Unifying Theories of Programming 2008, *Lecture Notes in Computer Science*, vol. 5713. Springer-Verlag (2010)
17. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing **15**(2 - 3), 146 — 181 (2003)
18. Cavalcanti, A.L.C., Woodcock, J.C.P.: A Tutorial Introduction to CSP in Unifying Theories of Programming. In: Refinement Techniques in Software Engineering, *Lecture Notes in Computer Science*, vol. 3167, pp. 220 – 268. Springer-Verlag (2006)
19. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering **SE-4**(3), 178 – 187 (1978)
20. Cristiá, M., Monetti, P.R.: Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing. In: K. Breitman, A.L.C. Cavalcanti (eds.) 11th International Conference on Formal Engineering Methods, *Lecture Notes in Computer Science*, vol. 5885, pp. 167 – 185. Springer-Verlag (2009)
21. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Formal Methods Europe, *Lecture Notes in Computer Science*, vol. 670, pp. 268 – 284. Springer-Verlag (1993)
22. Eertink, E.H.: Simulation techniques for the validation of LOTOS specifications. Ph.D. thesis, University of Twente (1994)
23. Fischer, C.: How to Combine Z with a Process Algebra. In: J. Bowen, A. Fett, M. Hinchey (eds.) ZUM'98: The Z Formal Specification Notation. Springer-Verlag (1998)
24. Fischer, C.: Combination and Implementation of Processes and Data: from CSP-OZ to Java. Ph.D. thesis, Fachbereich Informatik Universität Oldenburg (2000)
25. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: K. Havelund, M. Núñez, G. Rosu, B. Wolff (eds.) Formal Approaches to Software Testing and Runtime Verification, no. 4262 in Lecture Notes in Computer Science, pp. 40 – 54. Springer (2006)
26. Freitas, A.F., Cavalcanti, A.L.C.: Automatic Translation from *Circus* to Java. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) FM 2006: Formal Methods, *Lecture Notes in Computer Science*, vol. 4085, pp. 115 – 130. Springer-Verlag (2006)
27. Fujiwara, S., Bochmann, G.: Testing non-deterministic finite state machines with fault coverage. In: 4th International Workshop on Protocol Test Systems (1991)

28. Galloway, A.J.: Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-perspective Systems. Ph.D. thesis, University of Teeside, School of Computing and Mathematics (1996)
29. Gannon, J., McMullin, P., Hamlet, R.: Data abstraction implementation, specification and testing. ACM Transactions on Programming Languages and Systems **3**(3), 211–223 (1981)
30. Gaudel, M.C.: Testing can be formal, too. In: International Joint Conference, Theory And Practice of Software Development, *Lecture Notes in Computer Science*, vol. 915, pp. 82 – 96. Springer-Verlag (1995)
31. Gaudel, M.C., James, P.J.: Testing algebraic data types and processes : a unifying theory. Formal Aspects of Computing **10**(5-6), 436 – 451 (1998)
32. Helke, S., Neustupny, T., Santen, T.: Automating Test Case Generation from Z Specifications with Isabelle. In: J.P. Bowen, M.G. Hinchey, D. Till (eds.) International Conference of Z Users, *Lecture Notes in Computer Science*, vol. 1212, pp. 52 – 71. Springer-Verlag (1997)
33. Hennessy, M.C.B.: Algebraic Theory of Processes. MIT Press (1988)
34. Hierons, R., Sadeghipour, S., Singh, H.: Testing a system specified using statecharts and Z. Information and Software Technology **43**(2), 137 – 149 (2001)
35. Hierons, R.M.: Testing from a Z Specification. Software Testing, Verification & Reliability **7**, 19 – 33 (1997)
36. Hierons, R.M.: 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)
37. Hierons, R.M., Kim, T.H., Ural, H.: On the testability of SDL specifications. Computer Networks **44**(5), 681–700 (2004)
38. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
39. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall (1998)
40. Hoenick, J., Olderog, E.R.: Combining specification techniques for processes, data and time. In: M.J. Butler, L. Petre, K. Sere (eds.) Integrated Formal Methods, *Lecture Notes in Computer Science*, vol. 2335, pp. 245 – 266 (2002)
41. Kahsai, T., Roggenbach, M., Schlingloff, B.H.: Specification-based testing for refinement. In: SEFM '07: 5th IEEE International Conference on Software Engineering and Formal Methods, pp. 237 – 246. IEEE Computer Society (2007)
42. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE, vol. 84, pp. 1090 – 1126 (1996)
43. Lestiennes, G., Gaudel, M.C.: Testing processes from formal specifications with inputs, outputs, and datatypes. In: IEEE International Symposium on Software Reliability Engineering, pp. 3 – 14 (2002)
44. Lucio, L., Pedro, L., Buchs, D.: A Test Language for CO-OPN Specifications. In: 16th IEEE International Workshop on Rapid System Prototyping, pp. 195 – 201. IEEE Computer Society (2005)
45. Mahony, B., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering **26**(2), 150 – 177 (2000)
46. Martin, A.: Machine-Assisted Theorem-Proving for Software Engineering. Ph.D. thesis, Oxford Universiversity Computing Laboratory, Pembroke College, Oxford - UK (1995)
47. Massink, M., Latella, D., Gnesi, S.: On testing UML statecharts. The Journal of Logic and Algebraic Programming **69**(1-2), 1 – 74 (2006)
48. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
49. Nicola, R.D., Hennessy, M.C.B.: Testing equivalences for processes. Theoretical Computer Science **3**(1-2), 83 – 133 (1984)
50. Nogueira, S., Sampaio, A.C.A., Mota, A.C.: Guided test generation from csp models. In: J.S. Fitzgerald, A.E. Haxthausen, H. Yenigün (eds.) 5th International Colloquium on Theoretical Aspects of Computing, *Lecture Notes in Computer Science*, vol. 5160, pp. 258 – 273. Springer (2008)
51. Olderog, E., Wehrheim, H.: Specification and (property) inheritance in csp-oz. Science of Computer Programming **55**(1-3), 227–257 (2005)

52. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs Using *Circus*. Ph.D. thesis, University of York (2006)
53. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Formal development of industrial-scale systems. Innovations in Systems and Software Engineering **1**(2), 126 – 147 (2005)
54. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for *Circus*. Formal Aspects of Computing **21**(1-2), 3 – 32 (2009)
55. Peleska, J., Siegel, M.: Test automation of safety-critical reactive systems. In: Formal Methods Europe, Industrial Benefits and Advances in Formal Methods, *Lecture Notes in Computer Science*, vol. 1051 (1996)
56. Peleska, J., Siegel, M.: Test automation of safety-critical reactive systems. South African Computer Journal **19**, 53 – 77 (1997)
57. Péraire, C., Barbey, S., Buchs, D.: Test selection for object-oriented software based on formal specifications. In: W.P.d.R. D. Gries (ed.) Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods, *IFIP Conference Proceedings*, vol. 125, pp. 385 – 403. Chapman & Hall (1998)
58. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall (1998)
59. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley (2000)
60. Schoot, H.V.D., Ural, H.: Data flow oriented test selection for LOTOS. Computer Networks and ISDN Systems **27**(7), 1111 – 1136 (1993)
61. Sherif, A., Jifeng, H., Cavalcanti, A.L.C., Sampaio, A.C.A.: A framework for specification and validation of real-time systems using *circus* actions. In: Z. Liu, K. Araki (eds.) International Colloquium on Theoretical Aspects of Computing, *Lecture Notes in Computer Science*, vol. 3407, pp. 478 – 493. Springer-Verlag (2005)
62. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers (1999)
63. Srivatanakul, T., Clark, J.A., Stepney, S., Polack, F.: Challenging formal specifications by mutation: a CSP security example. In: 10th Asia-Pacific Software Engineering Conference, pp. 340 – 350. IEEE Press (2003)
64. Stoddart, B.: An Introduction to the Event Calculus. In: J.P. Bowen, M.G. Hinchey, D. Till (eds.) International Conference of Z Users, *Lecture Notes in Computer Science*, vol. 1212, pp. 52 – 71. Springer-Verlag (1997)
65. Taguchi, K., Araki, K.: The State-based CCS Semantics for Concurrent Z Specification. In: M. Hinchey, S. Liu (eds.) International Conference on Formal Engineering Methods, pp. 283 – 292. IEEE (1997)
66. Tang, X., Woodcock, J.C.P.: Towards Mobile Processes in Unifying Theories. In: 2nd International Conference on Software Engineering and Formal Methods – SEFM 2004, pp. 44 – 53. IEEE Computer Society (2004)
67. Tang, X., Woodcock, J.C.P.: Travelling Processes. In: D. Kozen, C. Shankland (eds.) Mathematics of Program Construction – MPC 2004, *Lecture Notes in Computer Science*, vol. 3125, pp. 381 – 399. Springer-Verlag (2004)
68. Treharne, H., Schneider, S.: Using a process algebra to control B OPERATIONS. In: 1st International Conference on Integrated Formal Methods – IFM'99, pp. 437 – 457. Springer-Verlag (1999)
69. Tretmans, J.: Test Generation with Inputs, Outputs, and Quiescence. In: Tools and Algorithms for the Construction and Analysis of Systems, *Lecture Notes in Computer Science*, vol. 1055, pp. 127 – 146. Springer-Verlag (1996)
70. Tripathy, P., Sarikaya, B.: Test Generation from LOTOS Specifications. IEEE Transactions on Computers **40**(4), 543–552 (1991)
71. Woodcock, J.C.P., Cavalcanti, A.L.C., Freitas, L.: Operational Semantics for Model-Checking *Circus*. In: J. Fitzgerald, I.J. Hayes, A. Tarlecki (eds.) FM 2005: Formal Methods, *Lecture Notes in Computer Science*, vol. 3582, pp. 237 – 252. Springer-Verlag (2005)
72. Woodcock, J.C.P., Cavalcanti, A.L.C., Gaudel, M.C., Freitas, L.J.S.: Operational Semantics for *Circus*. Formal Aspects of Computing To appear.
73. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)