Data Flow Coverage for *Circus*-Based Testing

Ana Cavalcanti[1] and Marie-Claude Gaudel[2]

[1] University of York, Department of Computer Science, York YO10 5GH, UK
[2] LRI, Université de Paris-Sud and CNRS, Orsay 91405, France

Abstract. *Circus* is a state-rich process algebra based on Z and CSP that can be used for testing. In this paper, we consider data-flow coverage. In adapting the classical results on coverage of programs to *Circus* models, we define a notion of specification traces, consider models with data-flow anomalies, and cater for the internal nature of state. Our results are a framework for data-flow coverage of such abstract models, a novel data-flow criterion suited to state-rich process models, and the conversion of specification traces into symbolic traces.

1 Introduction

The use of formal models, especially those underlying process algebra, as a basis for testing is now widely studied. *Circus* [5] is a very expressive and feature-rich algebra; it belongs to the important family of notations that combine the advantages of operational calculi like CSP [14] with specification languages like Z [17], thus comprising abstract data types at their core. For testing from such notations, it is appealing and natural to guide selection of test data from symbolically derived test traces using data-oriented criteria [13] that have been demonstrated to be good at detecting faults on data dependencies.

In previous work, we have defined a testing theory for *Circus* [2]. Following its operational semantics, this theory uses constrained symbolic traces: pairs formed by a symbolic trace and a constraint over the symbolic variables in the trace. Tests are built from such traces, enriched by observations (that is, refusals or acceptance) and verdict events; test sets that are exhaustive with respect to refinement in *Circus* have been defined. The constrained symbolic traces, however, capture the constraints raised by data operations and guards, but not their structure. We have, therefore, so far defined test-selection criteria based on notions like coverage of bounded symbolic traces or synchronisation coverage; information is missing to address data-flow coverage.

Here, we introduce specification traces, which include, besides communication events, internal data operations and guards. Based on these traces, we formalise notions of definitions, uses, and definition-clear paths for *Circus*. We define the conventional data-flow coverage criteria, and formalise a novel criteria inspired by [15] to cater for internal data flows. Finally, we consider how to construct constrained symbolic traces, and thus, symbolic tests from the specification traces, providing the link to the operational semantics. This result is relevant for all selection criteria based on specification traces (and not only data-flow criteria).

S. Gnesi and A. Rensink (Eds.): FASE 2014, LNCS 8411, pp. 415–429, 2014.

In summary, we present here the first collection of coverage criteria for *Circus* based on the structure of models. It is the first technique that takes advantage of the data model itself, rather than its semantics, in selecting tests. We prove unbias of the selected tests. This means that they cannot reject correct systems.

Data-flow coverage in the context of *Circus* requires adjustments. Firstly, data-flow anomalies must be accepted, because repeated definitions and definitions without use are routinely used in *Circus* abstract models. Second, due to the rich predicative data language of *Circus*, a concrete flow graph is likely much too big to be explicitly considered. Thus, tests are not based on paths of a flow graph, but on specification traces. Finally, the state of a *Circus* process is hidden, and so not all definitions and uses, and, therefore, not all data flows, are visible.

In the next section, we give an overview of the notations and definitions used in our work. Section 3 presents our framework, and Section 4, our new criterion. Section 5 addresses the general issue of constructing tests from selected specification traces. Finally, we consider related works in Section 6 and conclude in Section 7, where we also indicate lines for further work.

2 Background Material

This section describes *Circus*, its operational semantics, and data-flow coverage.

2.1 *Circus* Notation

A *Circus* model defines channels and processes like in CSP. Figure 1 presents an extract from the model of a cash machine. It uses a given set $CARD$ of valid cards, a set $Note$ of the kinds of notes available (10, 20, and 50), and a set $Cash == \mathrm{bag}\, Note$ to represent cash. The definitions of these sets are omitted.

The first paragraph in Figure 1 declares four channels: *inc* is used to request the withdrawal using a card of some cash, *outc* to return a card, *cash* to provide cash, and *refill* to refill the note bank in the machine. The second paragraph is an explicit definition for a process called *CashMachine*.

The first paragraph of the *CashMachine* definition is a Z schema *CMState* marked as the **state** definition. *Circus* processes have a private state, and interact with each other and their environment using channels. The state of *CashMachine* includes just one component: *nBank*, which is a function that records the available number of notes of each type: at most *cap*.

State operations can be defined by Z schemas. For instance, *DispenseNotes* specifies an operation that takes an amount $a?$ of money as input, and outputs a bag $notes!$ of *Note*s, if there are enough available to make up the required amount. *DispenseNotes* includes the schema $\Delta CMState$ to bring into scope the names of the state components defined in *CMState* and their dashed counterparts to represent the state after the execution of *DispenseNotes*. To specify $notes!$, we require that the sum of its elements ($\Sigma\, notes!$) is $a?$, and that, for each kind n of *Note*, the number of notes in $notes!$ is available in the bank. *DispenseNotes* also updates *nBank*, by decreasing its number of notes accordingly.

channel $inc : CARD \times \mathbb{N}_1$; $outc : CARD$; $cash : Cash$; $refill$

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

state $CMState == [\, nBank : Note \to 0 \,..\, cap \,]$

DispenseNotes

$\Delta CMState$
$a? : \mathbb{N}_1$; $notes! : Cash$

$\Sigma\, notes! = a?$
$\forall\, n : Note \bullet (notes! \sharp\ n) \leq nBank\ n \land nBank'\ n = (nBank\ n) - (notes! \sharp\ n)$

DispenseError

$\Xi CMState$
$a? : \mathbb{N}_1$; $notes! : Cash$

$\neg\ \exists\, ns : Cash \bullet \Sigma\, ns = a? \land \forall\, n : Note \bullet (ns \sharp\ n) \leq nBank\ n$
$notes! = [\![\ ]\!]$

$Dispense == DispenseNotes \lor DispenseError$

$$\bullet \left(\mu\ X \bullet \left(\begin{array}{l} inc?c?a \to \\ \quad X \\ \quad \sqcap\ outc!c \to X \\ \quad \sqcap \left(\begin{array}{l} \textbf{var}\ notes : Cash \bullet \\ \quad Dispense; \\ \quad \left(\begin{array}{l} \big(notes \neq [\![\ ]\!]\big)\ \&\ cash!notes \to \textbf{Skip} \\ \Box \\ \big(notes = [\![\ ]\!]\big)\ \&\ \textbf{Skip} \end{array} \right) \end{array} \right) ; outc!c \to X \\ \Box \\ refill \to (nBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\}\ ;\ X \end{array} \right) \right)$$

end

Fig. 1. Cash machine model

Another schema *DispenseError* defines the behaviour of the operation when there are not enough notes in the bank to provide the requested amount $a?$; the result is the empty bag $[\![\ ]\!]$. The Z schema calculus is used to define the total operation *Dispense* as the disjunction of *DispenseNotes* and *DispenseError*.

State operations are called actions in *Circus*, and can also be defined using Morgan's specification statements [11] or guarded commands from Dijkstra's language. CSP constructs can also be used to specify actions.

For instance, the behaviour of the process *CashMachine* is defined by a recursive action at the end after the '•'. A recursion $\mu\ X \bullet F(X)$ has a body given by $F(X)$, where occurrences of X are recursive calls. In our example, the recursion first offers a choice between an input $inc?c?a$, which accepts a card c and a request to withdraw the amount a, and a synchronisation on *refill*, which

is a request to fill the *nBank*. The actions that offer these communications are combined in an external choice ($\Box$) to be exercised by the environment.

If *refill* is chosen, an assignment changes the value of *nBank* to record a number *cap* of notes of all kinds. If $inc?c?a$ is chosen, then we have an internal (nondeterministic) choice of possible follow-on actions: recursing immediately (without returning the card or producing the money), returning the card via an output $outc!c$ before recursing, or considering the dispensation of cash before returning the card and recursing. In the dispensation, a local variable *notes* is declared, the operation *Dispense* is called, and then an external choice of two guarded actions is offered. If there is some cash available ($notes \neq [\![\ ]\!]$), then it can dispensed via $cash!notes$. Otherwise the action terminates (**Skip**). Here, nondeterminism comes from the fact that the specification does not go into details of bank management (stolen cards, bank accounts, and so on).

This example shows how Z and CSP constructs can be intermixed freely. A full account of *Circus* and its semantics is given in [12]. The *Circus* operational semantics is briefly discussed and illustrated in the next section.

2.2 *Circus* Operational Semantics and Tests

The *Circus* operational semantics [2] is distinctive in its symbolic account of state updates. As usual, it is based on a transition relation that associates configurations and a label. For processes, the configurations are processes themselves; for actions A, they are triples of the form $(\mathtt{c} \mid \mathtt{s} \models \mathtt{A})$.

The first component `c` of those triples is a constraint over symbolic variables used to define labels and the state. These are texts that denote *Circus* predicates (over symbolic variables). We use typewriter font for pieces of text. The second component `s` is a total assignment `x` := `w` of symbolic variables `w` to all state components `x` in scope. State assignments can also include declarations and undeclarations of variables using the constructs `var x := e` and `end x`. The state assignments define a specific value (represented by a symbolic variable) for all variables in scope. The last component of a configuration is an action `A`.

The labels are either empty, represented by ϵ, or symbolic communications of the form `c?w` or `c!w`, where `c` is a channel name and `w` is a symbolic variable that represents an input (?) or an output (!) value.

We define traces in the usual way. Due to the symbolic nature of configurations and labels, we obtain constrained symbolic traces, or *cstraces*, for short.

Example 1. Some of the cstraces of the process *CashMachine* are as follows.

$$(\langle\,\rangle, \mathtt{True}) \text{ and } (\langle \mathtt{refill}, \mathtt{inc}.\alpha_0.\alpha_1, \mathtt{outc}.\alpha_2 \rangle, \alpha_0 \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 = \alpha_0)$$

The first is the empty cstrace (empty symbolic trace with no constraint). The second records a sequence of interactions where a request for a *refill* is followed by a request for a withdraw of an amount α_1 using card α_0, followed by the return of a card α_2. The constraint captures those arising from the declaration of *inc*, namely, α_0 is a *CARD* and α_1, a positive number. It also captures the fact that the returned card is exactly that input ($\alpha_2 = \alpha_0$). $\Box$

As usual for process-algebra, tests of the *Circus* theory are constructed from traces. A cstrace defines a set of traces: those that can be obtained by instantiating the symbolic variables so as to satisfy the constraint. Accordingly, we have symbolic tests constructed from cstraces, and a notion of instantiation to construct concrete tests involving specific data. This approach is driven by the operational semantics of the language and led to the definition of symbolic exhaustive test sets and to proofs of their exhaustivity.

We observe that cstraces capture the constraints raised by data operations and guards, but not their structure.

Example 2. The following is a cstrace of *CashMachine* that captures a withdraw request followed by cash dispensation.

$$(\langle inc.\alpha_0.\alpha_1, cash.\alpha_2\rangle,\\ \alpha_0 \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \Sigma\,\alpha_2 = \alpha_1 \land \forall \mathtt{n} : \mathtt{Note} \bullet (\alpha_2 \ \sharp\ \mathtt{n}) \leq \mathtt{cap})$$

The constraint defines the essential properties of the cash α_2 dispensed, but not the fact that these properties are established by a variable declaration followed by a schema action call, and a guarded action. □

So, while cstraces are useful for trace-selection based on constraints, they do not support selection based on the structure of the *Circus* model. To this end, in [1] we have presented a collection of transition systems whose labels are pieces of the model: guards (predicates), communications, or simple *Circus* actions. The operational semantics for *Circus* defined by these transition systems is entirely compatible with the *Circus* original operational and denotational semantics, although it records information about the text of the model.

Thus, we use the transition relation $\Longrightarrow_{RP}$ from [1], written $\Longrightarrow$ here, to define a notion of specification traces, used to consider data-flow coverage criteria.

2.3 Data-Flow Coverage

Normally, the application of data-flow coverage criteria requires the absence of anomalies in the data-flow graph; this is not required or adequate here.

Data-flow coverage criteria were originally developed for sequential imperative languages based on the notion of definition-use associations [13]. They are traditionally defined in terms of a data-flow graph as triples (d, u, v), where d is a node in which the variable v is defined, that is, some value is assigned to it, u is a node in which the value of v is used, and there is a definition-clear path with respect to v from d to u. The strongest data-flow criterion, *all definition-use paths*, requires that, for each variable, every definition-clear path (with at most one iteration by loop) is executed. In order to reduce the number of tests required, weaker strategies such as *all-definitions* and *all-uses* have been defined.

When using these criteria, it is often assumed that there is no data-flow anomaly: on every path there is no use of a variable v not preceded by some node with a definition of v, and that after such a node, there is always some other node with a use of v [6]. These restrictions require preliminary checks and

facilitate the comparison of the criteria; they also ensure that there is always some test set satisfying the criteria. In *Circus*, anomalies lead to empty test sets.

Data-flow based testing in the case of abstract specifications with concurrency and communications requires adjustments (see, for instance [15] and Section 6) even if the notion of data-flow and the motivation are the same: to check dynamically data-flow dependencies via the execution of selected tests.

3 Data-Flow Coverage in *Circus*

Here, we define specification traces resulting from the transition relation $\Longrightarrow$, state the notions of definition and use of *Circus* variables, discuss anomalies, and present the definition of one of the classical coverage criteria.

3.1 Specification Traces

The main distinctive feature of the specification transition system in [1] is its labels. They record not only events, like in the operational semantics, but also guards and state changes. Moreover, they are expressed in terms of terms of the model, rather than symbolic variables. For example, for the *CashMachine*, we have labels `inc?c?a`, `var notes`, and `Dispense`. Finally, the specification-oriented system has no silent transitions, since they correspond to evolutions that are not guarded, and do not entail any communication or state change. These transitions do not capture observable behaviour, and so are not interesting for testing.

Like in the operational semantics, we have a transition relation $\Longrightarrow$ between texts of process. It is defined in terms of the corresponding relation for actions. For actions, a transition $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{g}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ establishes that in the state characterised by $(c_1 \mid s_1)$, if the guard g holds, then the next step in the execution of A_1 is the execution of A_2 in the state $(c_2 \mid s_2)$. Similarly, a transition $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{e}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ establishes that in the execution of $\mathtt{A}_1$ the event e takes place and then again the next step is the execution of A_2 in $(c_2 \mid s_2)$. Finally, $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{A}}{\Longrightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2)$ establishes that the first step is the action `A`, followed by A_2 in $(c_2 \mid s_2)$ and the remaining action to execute is A_2.

It is simple to define sequences of specification labels based on $\Longrightarrow$ and its associated transition relation $\Longrightarrow\!\!\!\!\!\!\Longrightarrow$ annotated with traces and defined as usual [3].

Example 3. For *CashMachine*, for instance, the following traces of specification labels, as well as their prefixes, are reachable according to $\Longrightarrow\!\!\!\!\!\!\Longrightarrow$.

$\langle \mathtt{inc?c?a}, \mathtt{outc!c}, \mathtt{inc?c?a}, \mathtt{var\ notes} \rangle$
$\langle \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} \neq [\![\ ]\!], \mathtt{cash!notes}, \mathtt{outc!c} \rangle$ □

We need, however, to consider enriched labels that include a tag to distinguish their various occurrences in the specification.

Example 4. In the traces in Example 3, the two occurrences of `outc!c` correspond to different occurrences of this piece of syntax in the model. Since we cannot consider repeated occurrences of labels to correspond to a single definition or use of a variable, we use tags to distinguish them. □

The tag can, for instance, be is related to the position of the labels in the model. We need a simple generalisation of the definition of $\Longrightarrow$, where a label is a pair containing a label (in the sense of Section 2.2) and a tag. We take the type *Tag* of tags as a given set, and do not specify a particular representation of tags.

For a process P, we define the set $sptraces(P)$ of sptraces of P: specification traces whose last label is observable, that is, a non-silent communication. This excludes traces that do not lead to new tests with respect to their prefixes.

Definition 1. *If we define* $obs(\mathtt{l}, t) \Leftrightarrow \mathtt{l} \in Comm \land \mathtt{l} \neq \epsilon$, *then we have*

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

$$sptraces(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1) = \{\mathtt{spt}, \mathtt{c}_2, \mathtt{s}_2, \mathtt{A}_2 \mid (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \stackrel{\mathtt{spt}}{\Rightarrow\!\!\!\!\!\!\Rightarrow} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \land spt \neq \langle\,\rangle \land obs(\mathrm{last}\ spt) \bullet \mathtt{spt}\}$$

Without loss of generality, we consider a process $\mathtt{begin\ state}[x : T] \bullet A\ \mathtt{end}$, with state components x of type T and a main action A. Its sptraces are those of $\mathtt{A}$, when considered in the state in which x has some value identified by the symbolic variable w_0, which is constrained to satisfy $w_0 \in T$. For actions A_1, the set $sptraces(\mathtt{c}_1, \mathtt{s}_1, \mathtt{A}_1)$ of its sptraces from the state characterised by the assignment s_1 and constraint c_1 is defined as those that can be constructed using $\stackrel{\mathtt{spt}}{\Rightarrow\!\!\!\!\!\!\Rightarrow}$ from the configuration $(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1)$ and whose last label is observable.

Example 5. Some sptraces of *CashMachine* are as follows. (In examples, we omit tags when they are not needed, and below we distinguish the two occurrences of `outc!c` by the tags `tag1` and `tag2`.)

$$\langle \mathtt{inc?c?a}, (\mathtt{outc!c}, \mathtt{tag1})\rangle \quad \langle \mathtt{inc?c?a}, (\mathtt{outc!c}, \mathtt{tag1}), \mathtt{inc?c?a}\rangle$$
$$\langle \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} \neq [\![\ ]\!], \mathtt{cash!notes}\rangle$$
$$\langle \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} \neq [\![\ ]\!], \mathtt{cash!notes}, (\mathtt{outc!c}, \mathtt{tag2})\rangle$$

We note that the first specification trace in Example 3 is not an sptrace. □

3.2 Definitions and Uses

In an sptrace, a definition is a tagged label, where the label is a communication or an action that may assign a new value to a *Circus* variable, that is, an input communication, a specification statement, a Z schema where some variables are written, an assignment, or a **var** declaration, which, in *Circus* causes an initialisation. The set $\mathrm{defs}(\mathtt{x}, \mathtt{P})$ of definitions of a variable x in a process P is defined in terms of the set $\mathrm{defs}(\mathtt{x}, \mathtt{spt})$ of definitions of x in a particular sptrace `spt`.

Definition 2. $\mathrm{defs}(\mathtt{x}, \mathtt{P}) = \bigcup \{\, \mathtt{spt} : sptraces(\mathtt{P}) \bullet \mathrm{defs}(\mathtt{x}, \mathtt{spt}) \,\}$

The set $\mathrm{defs}(\mathtt{x}, \mathtt{spt})$ can be specified inductively as follows.

Definition 3. $\mathrm{defs}(\mathtt{x}, \langle\,\rangle) = \varnothing$
$\mathrm{defs}(\mathtt{x}, \mathtt{tl} \frown \mathtt{spt}) = (\{\mathtt{tl}\} \cap \mathrm{defs}(\mathtt{x})) \cup \mathrm{defs}(\mathtt{x}, \mathtt{spt})$

The empty trace has no definitions. If the trace is a sequence formed by a tagged label $\mathtt{tl}$ followed by the trace $\mathtt{spt}$, we include $\mathtt{tl}$ if it is a definition of x as characterised by defs($\mathtt{x}$). The definitions of $\mathtt{spt}$ are themselves given by defs($\mathtt{x}, \mathtt{spt}$).

The tagged labels in which x is written (defined) can be specified as follows.

Definition 4. $\text{defs}(\mathtt{x}) = \{\, \mathtt{tl} : \mathit{TLabel} \mid \mathtt{x} \in \text{defV}(\mathtt{tl}) \,\}$

The set defV($\mathtt{tl}$) of such variables for a label $\mathtt{tl}$ is specified inductively; $\mathtt{g}$ stands for a guard, $\mathtt{d}$ for a channel, $\mathtt{e}$ an expression. The tags play no role here, and we ignore them in the definition below.

Definition 5

$$\text{defV}(\mathtt{g}) = \text{defV}(\epsilon) = \text{defV}(\mathtt{d}) = \text{defV}(\mathtt{d!e}) = \text{defV}(\mathtt{end\,y}) = \varnothing$$
$$\text{defV}(\mathtt{d?x}) = \text{defV}(\mathtt{d?x : c}) = \{\, \mathtt{x} \,\} \qquad \text{defV}(\mathtt{f} : [\mathtt{pre}, \mathtt{post}]) = \{\, \mathtt{f} \,\}$$
$$\text{defV}(\mathtt{Op}) = \mathit{wrtV}(\mathtt{Op}) \qquad \text{defV}(\mathtt{x} := \mathtt{e}) = \{\, \mathtt{x} \,\}$$
$$\text{defV}(\mathtt{var\,x : T}) = \{\, \mathtt{x} \,\} \qquad \text{defV}(\mathtt{var\,x} := \mathtt{e}) = \{\, \mathtt{x} \,\}$$

A Morgan specification statement $f : [\mathit{pre}, \mathit{post}]$ is a pre-post specification that can only modify the variables explicitly listed in the frame f.

The set $\mathit{wrtV}(\mathtt{Op})$ of written variables of a schema $\mathtt{Op}$ is defined in [5, page 161] as those that are potentially modified by Op, and their identification is not a purely syntactic issue. This set includes the state components v of Op that are not constrained by an equality $v' = v$. Following the usual over-approximation in data-flow analysis, we can take the pessimistic, but conservative, view that Op potentially writes to all variables in scope and avoid theorem proving.

We note that we are interested in variables, not channels. In an input $d?x$, the variable x is defined, but the particular channel d is not of interest. This reflects the fact that we are interested in the data flow, not the interaction specification.

Example 6. Coming back to the *CashMachine* (and ignoring tags) we have:

$$\begin{aligned}
&\text{defs}(\mathtt{c}, \mathtt{CashMachine}) = \{\mathtt{inc?c?a}\}\\
&\text{defs}(\mathtt{a}, \mathtt{CashMachine}) = \{\mathtt{inc?c?a}\}\\
&\text{defs}(\mathtt{notes}, \mathtt{CashMachine}) = \{\mathtt{var\ notes} : \mathtt{Cash}, \mathtt{Dispense}\}\\
&\text{defs}(\mathtt{nBank}, \mathtt{CashMachine}) = \{\, \mathtt{Dispense},\\
&\qquad\qquad \mathtt{nBank} := \{\, 10 \mapsto \mathtt{cap}, 20 \mapsto \mathtt{cap}, 50 \mapsto \mathtt{cap} \,\}\}
\end{aligned}$$

□

The notion of (externally visible) use is simpler: a tagged label with an output communication. Formally, the set e-uses($\mathtt{x}, \mathtt{P}$) of uses of a variable x in a process P can be identified from its set of sptraces.

Definition 6. $\text{e-uses}(\mathtt{x}, \mathtt{P}) = \bigcup \{\, \mathit{spt} : \mathit{sptraces}(\mathtt{P}) \bullet \text{e-uses}(\mathtt{x}, \mathtt{spt}) \,\}$

The set e-uses($\mathtt{x}, \mathtt{spt}$) of uses of x in a trace $\mathtt{spt}$ can be specified as follows.

Definition 7. $\text{e-uses}(\mathtt{x}, \langle\,\rangle) = \varnothing$

$$\text{e-uses}(\mathtt{x}, \mathtt{tl} \frown \mathtt{spt}) = (\{\mathtt{tl}\} \cap \text{e-uses}(\mathtt{x})) \cup \text{e-uses}(\mathtt{x}, \mathtt{spt})$$

Finally, uses of a variable x are labels $(\mathtt{d!e}, t)$ where x occurs free in the expression e. $FV(\mathtt{e})$ denotes the set of free variables of an expression $\mathtt{e}$.

Definition 8. $\text{e-uses}(\mathtt{x}) = \{\mathtt{d} : \mathit{CName};\ \mathtt{e} : \mathit{Exp};\ t : \mathit{Tag} \mid \mathtt{x} \in FV(\mathtt{e}) \bullet (\mathtt{d!e}, t)\}$

At this point, we consider e-uses, but not the classical notion of p-uses, which relates to uses in predicates and, in the context of *Circus*, are not observable. We introduce a notion of internal uses (i-uses) later on in Section 4.1.

Example 7. We have $\text{e-uses}(\mathtt{c}, \mathtt{CashMachine}) = \{(\mathtt{outc!c}, \mathtt{tag1}), (\mathtt{outc!c}, \mathtt{tag2})\}$ and $\text{e-uses}(\mathtt{notes}, \mathtt{CashMachine}) = \{\mathtt{cash!notes}\}$. There are no other externally visible uses in *CashMachine*. □

We observe that a label cannot be both a definition and a use of a variable, because a use is an output communication, which does not define any variable. Besides, a label can be neither a definition nor a use (this is the case for `refill`) and then not considered for data-flow coverage.

The property $\text{clear-path}(\mathtt{spt}, \mathtt{df}, \mathtt{u}, \mathtt{x})$ characterises the fact that the trace `spt` has a subsequence that starts with the label `df`, finishes with the label `u`, and has no definition of the variable x. (We consider subsequences of a trace, but, for consistency with classical terminology, we use the term path anyway.)

Definition 9

$$\begin{aligned}\text{clear-path}(\mathtt{spt}, \mathtt{df}, \mathtt{u}, \mathtt{x}) \Leftrightarrow\ & \exists\, i : 1 \mathinner{..} \#\, spt \bullet spt\ i = \mathtt{df} \land \\ & \exists\, j : (i+1) \mathinner{..} \#\, spt \bullet spt\ j = \mathtt{u} \land \\ & \forall\, k : (i+1) \mathinner{..} (j-1) \bullet spt\ k \notin \text{defs}(\mathtt{x}, \mathtt{P})\end{aligned}$$

A e-use `u` of a variable x is said to be reachable by a definition `df` of x if there is a trace `spt` such that $\text{clear-path}(\mathtt{spt}, \mathtt{df}, \mathtt{u}, \mathtt{x})$.

3.3 Data-Flow Anomalies and *Circus*

Three data-flow anomalies are usually identified: (1) a use of a variable without a previous definition; (2) two definitions without an intermediate use; and (3) a definition without use. While these all raise concerns in a program, it is not the case in a *Circus* model. Because a variable declaration is a variable definition that assigns an arbitrary value to a variable, it is common to follow it up with a second definition that restricts that value.

In addition, it is not rare to use a communication `d?x` to define just that the value x to be input via the channel `d` is not restricted (and also later not used). In an abstract specification, a process involving such a communication might, for example, be combined in parallel with another process that captures another requirement concerned with restricting these values x, while the requirement captured by the process that defines `d?x` is not concerned with such values.

For the data-coverage criteria that we consider, when a definition involved in any of the above anomalies is considered, it imposes no restriction on the set of tests under consideration for coverage. In practical terms, no tests are required.

3.4 All-Defs

The data-coverage criterion that we present here, all-defs, requires that all definitions are covered, and followed by one (reachable) use, via any (clear) path. We formalise coverage criterion by identifying the sets of sptraces *SSPT* that satisfy that criterion. For all-defs, the formal definition is as follows.

Definition 10. *For every variable name x and process P, a set SSPT of sptraces of P provides all-defs coverage if, and only if,*

$$\begin{array}{l}\forall\, \mathtt{df} : \text{defs}(\mathtt{x}, \mathtt{P}) \bullet \\ \quad (\exists\, \mathtt{spt} : \mathit{sptraces}(\mathtt{P});\ \mathtt{u} : \text{e-uses}(\mathtt{x}, \mathtt{P}) \bullet \text{clear-path}(\mathtt{spt}, \mathtt{df}, \mathtt{u}, \mathtt{x})) \Rightarrow \\ \quad\quad (\exists\, \mathtt{spt} : \mathit{SSPT};\ \mathtt{u} : \text{e-uses}(\mathtt{x}, \mathtt{P}) \bullet \text{clear-path}(\mathtt{spt}, \mathtt{df}, \mathtt{u}, \mathtt{x}))\end{array}$$

If there is an sptrace that can contribute to coverage, then at least one is included.

Example 8. As previously explained, in *CashMachine*, inc?c?a is the only definition of c, and its two uses are (outc!c, tag1) and (outc!c, tag2). Examples of sets of sptraces that provide all-defs coverage are the three singletons below.

$$\begin{array}{l}\{\langle \mathtt{inc?c?a}, (\mathtt{outc!c}, \mathtt{tag1})\rangle\} \\ \{\langle \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} = [\![\ ]\!], (\mathtt{outc!c}, \mathtt{tag2})\rangle\} \\ \{\langle \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} \neq [\![\ ]\!], \mathtt{cash!notes}, (\mathtt{outc!c}, \mathtt{tag2})\rangle\}\end{array}$$

Other sets that provide all-defs coverage are the supersets of the above sets, and the sets that include any of the extensions of the sptraces above. □

In [3] we define the classical all-uses and all-du-paths criteria.

The *CashMachine* variables *nBank* and *a* are used internally only. There is no clear path from their definition to an external use, and so every set of sptraces provides coverage (according to all-defs and the other classical criteria) with respect to these variables. They contribute, however, to our next criterion.

4 sel-var-df-chain-Trace

This criterion is based on the notion of a var-df-chain. The idea is to identify sptraces that include chains of definition and associated internal uses of variables, such that each variable affects the next one in the chain. Given the characteristics of *Circus*, it is very likely that most specifications contain a number of such chains.

4.1 var-df-chain

A suffix of an sptrace spt starting at position i (that is, $(i \,..\, \#\, spt) \upharpoonright spt$) is in the set var-df-chain(x, P) of var-df-chains of P for x if it starts with a label $spt\, i$ that defines x and subsequently has a clear path to a label $spt\, j$. This label must either be a use of x, and in this case it must be the last label of spt, or affect the definition of another variable y, and in this case spt must continue with a var-df-chain for y. The continuation is $(j \,..\, \#\, spt) \upharpoonright spt$, the suffix of spt from j.

Definition 11

$$\begin{array}{l}\text{var-df-chain}(\mathtt{x},\mathtt{P}) = \\ \quad \{\ \mathtt{spt} : \mathit{sptraces}(P);\ i : 1\,..\,\#\,\mathit{spt};\ j : (i+1)\,..\,\#\,\mathit{spt} \mid \\ \qquad \left(\begin{array}{l} \mathit{spt}\ i \in \text{defs}(\mathtt{x},\mathtt{P}) \land (\forall\, k : (i+1)\,..\,(j-1) \bullet \mathit{spt}\ k \notin \text{defs}(\mathtt{x},\mathtt{P})) \land \\ \left(\begin{array}{l} (\mathit{spt}\ j \in \text{e-uses}(\mathtt{x},\mathtt{P}) \land j = \#\,\mathit{spt}) \lor \\ (\exists\, \mathtt{y} \bullet \text{affects}(\mathtt{x},\mathtt{y},\mathit{spt}\ j) \land (j\,..\,\#\,\mathit{spt}) \upharpoonright \mathit{spt} \in \text{var-df-chain}(\mathtt{y},\mathtt{P})) \end{array}\right) \end{array}\right) \\ \qquad \bullet\ (i\,..\,\#\,\mathit{spt}) \upharpoonright \mathit{spt} \\ \quad \}\end{array}$$

A variable x affects the definition of another variable y in a tagged label `tl` if it is an internal use of x and a definition of y.

Definition 12. $\text{affects}(\mathtt{x},\mathtt{y},\mathtt{tl}) = \mathtt{x} \in \text{i-useV}(\mathtt{tl}) \land \mathtt{y} \in \text{defs}(\mathtt{tl})$

The set $\text{i-useV}(\mathtt{tl})$ of variables used internally in `tl` is defined as follows.

Definition 13

$$\begin{array}{ll} \text{i-useV}(\mathtt{g}) = FV(\mathtt{g}) & \text{i-useV}(\epsilon) = \text{i-useV}(\mathtt{d}) = \varnothing \\ \text{i-useV}(\mathtt{d!e}) = \text{i-useV}(\mathtt{d?x}) = \varnothing & \text{i-useV}(\mathtt{d?x} : \mathtt{c}) = FV(\mathtt{c}) \setminus \{x\} \\ \multicolumn{2}{l}{\text{i-useV}(\mathtt{f} : [\mathtt{pre},\mathtt{pos}]) = FV(\mathtt{pre}) \cup FV(\mathtt{pos})} \\ \text{i-useV}(\mathtt{Op}) = FV(\mathtt{Op}) & \text{i-useV}(\mathtt{x} := \mathtt{e}) = FV(\mathtt{e}) \\ \text{i-useV}(\mathtt{var\,x} : \mathtt{T}) = \varnothing & \text{i-useV}(\mathtt{var\,x} := \mathtt{e}) = FV(\mathtt{e}) \\ \text{i-useV}(\mathtt{end\,y}) = \varnothing & \end{array}$$

This notion of internal use subsumes the classical notion of p-uses.

4.2 The Criterion

We observe that var-df-chains are not sptraces, but suffixes of sptraces. So, coverage is provided by sptraces that have such suffixes, rather than by the var-df-chains themselves. In particular, sel-var-df-chain-trace coverage requires that every chain in a model is covered by at least one sptrace.

Definition 14. *For every variable name x and process P, a set SSPT of sptraces of P provides sel-var-df-chain-trace coverage if, and only if,*

$$\begin{array}{l} \forall\, \mathtt{spt_1} : \text{var-df-chain}(\mathtt{x},\mathtt{P}) \bullet \\ \quad \exists\, \mathtt{spt_2} : \mathit{SSPT};\ \mathtt{spt_3} : \text{seq}\ \mathit{TLabel} \bullet \mathit{spt_2} = \mathit{spt_3} \frown \mathit{spt_1} \end{array}$$

The specification trace $\mathtt{spt_3}$ is an initialisation trace that leads to the chain.

This criterion is the most demanding of the data-flow criteria defined in [3] where a formal proof of this result is available.

Example 9. The very basic var-df-chains, where the same variable is considered as the starting definition and the final use, with a clear path with respect to this variable in between, are covered by the classical all-du-paths criterion.

Table 1. Operational semantics of sptraces; $\mathtt{w}_0$ stand for fresh symbolic variables

$$\frac{c \wedge (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models \langle \mathtt{g} \rangle \frown \mathtt{spt}) \longrightarrow^{\epsilon}_{ST} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models \mathtt{spt})}$$

$$\frac{c \wedge T \neq \varnothing}{(\mathtt{c} \mid \mathtt{s} \models \langle \mathtt{d?x} : \mathtt{T} \rangle \frown \mathtt{spt}) \longrightarrow^{\mathtt{d?w}_0}_{ST} (\mathtt{c} \wedge \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\ x} := \mathtt{w}_0 \models \mathtt{spt})}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \langle \mathtt{d!e} \rangle \frown \mathtt{spt}) \longrightarrow^{\mathtt{d!w}_0}_{ST} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{w}_0 = \mathtt{e}) \mid \mathtt{s} \models \mathtt{spt})}$$

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{Skip})}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \langle \mathtt{A}_1 \rangle \frown \mathtt{spt}) \longrightarrow^{\epsilon}_{ST} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{spt})}$$

The label $\mathtt{nBank} := \{ 10 \mapsto \mathtt{cap}, 20 \mapsto \mathtt{cap}, 50 \mapsto \mathtt{cap} \}$ is such a definition, and *nBank* is used in `Dispense`. Moreover, *notes* is externally used in the label `cash!notes`. This leads to the following var-df-chain.

$$\langle \mathtt{nBank} := \{ 10 \mapsto \mathtt{cap}, 20 \mapsto \mathtt{cap}, 50 \mapsto \mathtt{cap} \},\ \mathtt{inc?c?a}, \mathtt{var\ notes}, \mathtt{Dispense}, \mathtt{notes} \neq [\![\ ]\!], \mathtt{cash!notes} \rangle$$

Its coverage leads to coverage of the effect of a *refill*, after which the value of *nBank* is updated. An initialisation trace for the above var-df-chain is $\langle \mathit{refill} \rangle$.

5 Conversion of Specification Traces to Symbolic Traces

Converting an sptrace to a symbolic trace requires an operational semantics for sptraces, which we provide in Table 1. It defines a transition relation $\longrightarrow_{ST}$ using four rules: one for when the first label is a guard, two for when it is either an input or an output, and one for an action label `A`. In this last case, the rules of the operational semantics transition rule $\longrightarrow$ define the new transition relation.

Like in the operational semantics, the configuration is a triple, but here, we have an sptrace associated with a constraint and a state assignment. From a configuration $(\mathtt{c} \mid \mathtt{s} \models \langle \mathtt{l} \rangle \frown \mathtt{spt})$ we have a transition to a configuration with `spt`. The new constraint and state depend on the label `l`.

For a guard, a transition requires that c is satisfiable and g holds in the current state $(s;\ g)$. In this case, the transition is silent: it has label ϵ.

Input and output communications give rise to non-silent transitions with labels that are symbolic inputs and outputs. Inputs `d?x: T` are annotated with the type T of channel d. The new constraint records that the input value represented by the fresh symbolic variable w_0 has type T and the state is enriched with a declaration of x whose initial value is set to w_0.

Finally, we have a transition relation $\xrightarrow{\mathtt{st}}$ that defines a symbolic trace *st* that captures the interactions corresponding to an sptrace. It is defined from $\longrightarrow_{ST}$

in the usual way [3], and used below to define the function $\text{cstrace}_{\text{SPT}}{}^{\mathtt{a}}(P)$ that characterises the set of cstraces of P in terms of $sptraces(P)$. The parameter $\mathtt{a}$ is an alphabet: a sequence of fresh symbolic variables. The cstraces defined by $\text{cstrace}_{\text{SPT}}{}^{\mathtt{a}}(P)$ use these variables in the order determined by $\mathtt{a}$.

Definition 15

$$\text{cstrace}_{\text{SPT}}{}^{\mathtt{a}}(\mathtt{begin\ state}[x : T] \bullet A\ \mathtt{end}) = \text{convSPT}^{\mathtt{a}}(\mathtt{w_0} \in \mathtt{T}, \mathtt{x} := \mathtt{w_0})\ (\!|\ sptraces(\mathtt{begin\ state}[x : T] \bullet A\ \mathtt{end})\ |\!)$$

As before, we consider a process $\mathtt{begin\ state}[x : T] \bullet A\ \mathtt{end}$ and define its cstraces by applying a conversion function $\text{convSPT}^{\mathtt{a}}(\mathtt{c}, \mathtt{s})$ to each of its sptraces.

Definition 16. *For every alphabet* $\mathtt{a}$*, constraint* $\mathtt{c}$*, state assignment* $\mathtt{s}$ *and sptrace* $\mathtt{spt}$*, we have that* $\text{convSPT}^{\mathtt{a}}(\mathtt{c}, \mathtt{s})\,\mathtt{spt} = (\mathtt{st}, \exists(\alpha\mathtt{c} \setminus \alpha\mathtt{st}) \bullet \mathtt{c_1})$ *where* $\mathtt{st}$ *and* $\mathtt{c_1}$ *are characterised by* $\alpha\mathtt{st} \leq \mathtt{a} \land \exists\, \mathtt{s_1} \bullet (\mathtt{c} \mid \mathtt{s} \models \mathtt{spt}) \xrightarrow{\mathtt{st}} (\mathtt{c_1} \mid \mathtt{s_1} \models \langle\,\rangle)$.

Each sptrace gives rise to exactly one cstrace, since any nondeterminism in the actions is captured by the constraint on the symbolic variables. The alphabet $\alpha\mathtt{st}$ of the symbolic trace st is a prefix of a: $\alpha\mathtt{st} \leq \mathtt{a}$. We note that convSPT is a linear translation and can be implemented with a good computational complexity compared to the test cases generation itself.

Example 10. The following cstraces correspond to the sptraces in Example 8.

$$(\langle \mathtt{inc}?\alpha_0?\alpha_1, \mathtt{outc}!\alpha_2\rangle, \alpha_0 \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 = \alpha_0)$$
$$(\langle \mathtt{inc}?\alpha_0?\alpha_1, \mathtt{cash}!\alpha_2, \mathtt{outc}!\alpha_3\rangle, \alpha_0 \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \Sigma\alpha_2 = \alpha_1 \land (\exists\, \mathtt{w_0} : \mathtt{Note} \to \mathbb{N} \bullet (\forall \mathtt{n} : \mathtt{Note} \bullet \alpha_2\ \sharp\ \mathtt{n}) \leq \mathtt{w_0\,n})) \land \alpha_3 = \alpha_0)$$

We take the alphabet to be $\langle \alpha_0, \alpha_1, \ldots \rangle$. The first cstrace comes from both the first and the second sptrace. The second cstrace comes from the last sptrace. The symbolic variable $\mathtt{w_0}$ denotes the internal value of *nBank*, which is not observable in the trace, but contributes to the specification of the observable value α_2.

Two sptraces give rise to the same cstrace because after a withdraw request, the card may be returned immediately for one of two reasons: a problem with the card account (like insufficient funds) or no money in the cash machine. Since the model abstracts away the existence of accounts, they cannot be distinguished by tests from this model. This is reflected in the fact that the two sptraces have different tags associated with $outc!c$. This indicates that they correspond to two different occurrences in the model. □

Contrary to the cstraces defined by the operational semantics, which capture just observable labels, sptraces are defined specifically to capture the structure of the model, and thus guards and data operations that may not be visible in the interface of the SUT. So, it is not surprising that there are sptraces that lead to the same cstrace. They come from paths in the model that are not distinguishable by observing the SUT. Requiring their absence in programs is reasonable, but abstract specifications may lead to such situations. It is not an issue for test

generation, but it may be a problem for understanding or observing the SUT when running the tests. A test generation tool might, for example, warn that a distinction may need to be introduced, or instrumented, in the SUT.

The next theorem establishes that tests identified by sptraces are unbiased with respect to the operational semantics because they specify valid cstraces of the process. Construction of unbiased tests from cstraces was addressed in [2].

Theorem 1. $\mathrm{cstrace}_{\mathrm{SPT}}{}^{\mathsf{a}}(\mathsf{P}) \subseteq \mathit{cstraces}^{\mathsf{a}}(\mathsf{P})$

We do not have equality: there is no empty sptrace, for instance. A proof is in [3]. The main lemma is proved by induction on the specification traces of P.

6 Related Works

Data-flow based testing for state-based specification languages has been applied to Lotos [15], to SDL and Estelle (that is, EFSM) [16], and extended with control dependencies in [8]. Our sel-var-df-chain-trace selection criterion is inspired from [15], but different, due to the notion of internal state in *Circus* and to the forms of symbolic tests considered in the *Circus* testing theory (see [3] for details). These differences, however, should not prevent its extension to control dependencies, possibly by some slight enrichment of our tagged labels.

In another context, Tse et al. have adapted data-flow testing to service orchestrations specified in WS-BPEL in [9], and to service choreographies in [10]. From the specifications, they build an XPath Rewriting Graph, which captures the specificities of the underlying process algebra, which is very different from *Circus*, with loose coupling between processes, XML messages, and XPath queries.

Testing tools based on symbolic input-output transition systems, and a symbolic version of the *ioco* conformance relation have been presented by Clarke et al. in [4] and by Frantzen et al. [7]. The models and relations are different from ours, since there is a semantic distinction between inputs and outputs, no data structures, and no hidden state. In both [4] and [7], test selection is based on test purposes. In [7], there is a similar notion of symbolic traces, with a formula constraining the interaction variables of the trace, and another constraint on the update of the state variables. Data-flow coverage, however, was not addressed and is less relevant than for *Circus* given the limited operations on data.

7 Conclusions

We have presented a framework for test selection from *Circus* models based on data-flow coverage criteria for specification traces, which record sequences of guards, communications and actions of a model. Using these definitions, we have formalised some coverage criteria, including a new criterion that takes into account internal definitions and uses. Proof of unbias of the selected tests is possible due to formal nature of our setting. We have formalised also the construction of cstraces (used to construct tests) from specification traces.

The specification traces defined in this paper can be used for other selection criteria, data-flow based and other ones as well, since most features of the models are kept. On these bases, it is our plan to consider a number of selection criteria for *Circus* tests, and to explore criteria that consider a variety of *Circus* constructs in an integrated way, to include, for instance, notions of Z schema coverage, case splitting in the pre and postcondition of specification statements, control dependencies and test purposes expressed in *Circus*. We plan also to address in a formal framework the problem of monitoring such tests.

Acknowledgments. We warmly thank Frédéric Voisin and referees for several pertinent comments. We are grateful to the Royal Society and the CNRS for funding our collaboration.

References

1. Cavalcanti, A., Gaudel, M.-C.: Specification Coverage for Testing in *Circus*. In: Qin, S. (ed.) UTP 2010. LNCS, vol. 6445, pp. 1–45. Springer, Heidelberg (2010)
2. Cavalcanti, A.L.C., Gaudel, M.-C.: Testing for Refinement in *Circus*. Acta Informatica 48(2), 97–147 (2011)
3. Cavalcanti, A.L.C., Gaudel, M.-C.: Data Flow Coverage of *Circus* Specifications - extended version. RR 1567, LRI, Univ. Paris-Sud XI (December 2013), https://www.lri.fr/ bibli/Rapports-internes/2013/RR1567.pdf
4. Clarke, D., Jéron, T., Rusu, V., Zinovieva, E.: STG: A Symbolic Test Generation Tool. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 470–475. Springer, Heidelberg (2002)
5. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. FACJ 15(2-3), 146–181 (2003)
6. Clarke, L.A., Podgurski, A., Richardson, D.J., Zeil, S.J.: A Comparison of Data Flow Path Selection Criteria. In: ICSE, pp. 244–251 (1985)
7. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006/RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
8. Hong, H.S., Ural, H.: Dependence testing: Extending data flow testing with control dependence. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 23–39. Springer, Heidelberg (2005)
9. Mei, L., Chan, W.K., Tse, T.H.: Data flow testing of service-oriented workflow applications. In: ICSE, pp. 371–380 (2008)
10. Mei, L., Chan, W.K., Tse, T.H.: Data flow testing of service choreography. In: ESEC/FSE, pp. 151–160 (2009)
11. Morgan, C.C.: Programming from Specifications, 2nd edn. Prentice-Hall (1994)
12. Oliveira, M.V.M., Cavalcanti, A.L.C., Woodcock, J.C.P.: A UTP Semantics for *Circus*. FACJ 21(1-2), 3–32 (2009)
13. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE TSE 11(4), 367–375 (1985)
14. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2011)
15. Schoot, H.V.D., Ural, H.: Data flow analysis of system specifications in LOTOS. Int. Journal of Software Engineering and Knowledge Engineering 7, 43–68 (1997)
16. Ural, H., Saleh, K., Williams, A.W.: Test generation based on control and data dependencies. Computer Communications 23(7), 609–627 (2000)
17. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)