A note on traces refinement and the *conf* relation in the Unifying Theories of Programming

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

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

Abstract. There is a close relation between the failures-divergences and the UTP models of CSP, but they are not equivalent. For example, miracles are not available in the failures-divergences model; the UTP theory is richer and can be used to give semantics to data-rich process algebras like *Circus*. Previously, we have defined functions that calculate the failures-divergences model of a CSP process characterised by a UTP relation. In this note, we use these functions to calculate the UTP characterisations of traces refinement and of the *conf* relation that is widely used in testing. In addition, we prove that the combination of traces refinement and *conf* corresponds to refinement of processes in *Circus*. This result is the basis for a formal testing technique based on *Circus*; as usual in testing, we restrict ourselves to divergence-free processes.

1 Introduction

Formal specifications have been widely explored as a starting point for software testing; the works reported in [8, 11, 3, 4, 2, 14] give a few examples. In our own previous work [5], we have instantiated Gaudel's long-standing theory of formal testing [12] to CSP [19]. We now face the challenge of a richer language: *Circus* [6], which combines CSP with Z [20] and Morgan's refinement calculus [17] to provide a notation that supports refinement of reactive systems with state.

The *Circus* semantic model [18] is based on the UTP [13]; a *Circus* process is characterised by a relation in a restriction of the UTP theory for CSP. In this model, we can define, for example, the application of CSP constructs like external choice to processes that involve operations on a local state. We can also accommodate miraculous specifications from Morgan's refinement calculus.

In previous work, to study the relationship between the UTP and the canonical failures-divergences model of CSP, we have defined functions that calculate the failures-divergences model of a UTP relation that characterises a CSP process [7]. The UTP theory for CSP is richer than the failures-divergences model. In addition, refinement in the UTP is in close correspondence with failures-divergences refinement, but the other two main refinement relations that support compositional and stepwise reasoning in CSP, namely traces and failures refinement, have not been studied in the UTP.

Traces refinement is useful for reasoning about safety properties of a process; it ensures that the implementation does not engage in any interactions with the environment that are not allowed by the specification. Failures refinement, on the other hand, is used for reasoning about liveness: the interactions in which an implementation has to be prepared to engage.

Our interest is in a testing technique based on *Circus* specifications; our long-term goal is to provide automated support for test generation with the objective of verifying that a system under test implements a *Circus* specification correctly. In other words, we are interested in testing for refinement using *Circus*.

It is usual for testing techniques to assume that the specification and the system under test are divergence free. In addition, testing for trace inclusion and for reduction of deadlock is typically carried out separately to simplify the individual tests. Trace inclusion is, of course, traces refinement in the context of CSP, and reduction of deadlock, is captured by a relation usually called *conf* (for conformance). For CSP, we have proved that traces refinement and *conf*, together, are equivalent to failures-divergences refinement for divergence-free processes.

For *Circus*, we follow a similar approach, but there is no accepted definition of traces refinement and *conf* in the UTP. In this note, we calculate definitions for these relations, and prove that their combination corresponds to refinement of divergence-free *Circus* processes. Our calculations are based on functions that map UTP relations to components of the failures-divergences model. A perhaps surprising result is that the combination of traces refinement and *conf* do not correspond to the refinement relation in the UTP, but to refinement when state components are encapsulated; this is the notion of process refinement in *Circus*.

In the next section, we discuss the requirements and assumptions that are common to testing techniques based on process algebra. Afterwards, in Section 3, we give a brief and informal presentation of our process algebra of choice: *Circus*. Sections 4, 5, and 6 present our main results: the calculations of UTP characterisations of traces refinement and *conf*, and a proof that, together, they correspond to process refinement. Finally, in Section 7, we summarise our results and discuss our plans for future work. An appendix presents a few lemmas used in the proofs of our main theorems.

2 Process-algebra based formal testing

In this section we briefly recall some basic principles of specification-based testing.

In testing, an executable system, called the *system under test (SUT)* is given as a black-box. We can only observe the behavior of the SUT on any chosen input, or input sequence, and then decide whether it is acceptable or not with respect to some description of its intended behavior.

Given a formal specification SP and an SUT, any testing activity is, explicitly or not, based on a satisfaction relation: SUT *sat* SP. Since the SUT is a black-box, the testing process consists in using the specification SP to construct a set of tests, such that the SUT passes them if, and only if, it satisfies SP.

The tests are derived from the specification on the basis of the satisfaction relation, and often on the basis of some additional knowledge of the SUT and of its operational environment called *testability hypothesis*. Such test sets are called *exhaustive* in [12] or *complete* by other authors [4].

In the case of specifications based on some process algebra, tests are processes built on the same alphabet of events as the specification (possibly enriched by some special symbols). The execution of a given test consists in running it and the SUT in parallel. This can be done under the assumption (testability hypothesis) that the SUT behaves as some unknown, maybe infinite, transition system.

This testability hypothesis builds a bridge between the notions of satisfaction, (as introduced above, between a system and a specification) and refinement between two models: a specification model and an implementation model. Refinement has the advantage of being formalisable and well studied, while satisfaction is less easily formalisable, since it relates a model and a system.

The verdict about the success or not of a test execution depends on the observations that can be made, and it is based on the satisfaction relation. Most testing methods based on process algebras consider that two kinds of observations are possible: external events, and deadlock (that is, refusal of some external events). Deadlock is observed via time-out mechanisms: it is assumed that if the SUT does not react after a given time limit, it is blocked.

Divergences raise problems of observability; generally, it is not possible to distinguish a divergent from a deadlocked system using testing. So, most methods assume that the SUT is divergence free. This is equivalent to identifying divergence with deadlock in the unknown models of the systems under test; most authors, including us in [5], circumvent the problem of observability in this way. If the SUT is divergent, the divergence is detected as a (probably forbidden) deadlock and reported as such by the verdict of the tests.

Exhaustive test sets are often infinite, or too large to be used in practice. They are, however, used as references for selecting finite, practical, test subsets according to a variety of criteria, such as additional hypotheses on the SUT [2], coverage of the specification [8, 14], or test purposes [10].

3 *Circus*

A *Circus* program is a sequence of paragraphs just like in Z, but we can declare channels and processes. A system is specified in *Circus* as a process: it encapsulates a state, and exhibits some behaviour. Figure 1 gives a small example: the specification of a fresh identifier generator; it has four paragraphs. The first paragraph declares a given set ID containing all valid identifiers. The second and third paragraphs declare a few channels: *req* is used to request a fresh identifier, which is output by the system using the channel *out*; and the channel *ret* is used to return an identifier that is no longer required. The type of a channel determines the values that it can communicate; in the case of *req*, the absence of a type declaration indicates that it is used only for synchronisation.

$[ID]$

channel req

channel $ret, out : ID$

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

    state $S == [\, idS : \mathbb{P}\, ID \,]$

    $Init \mathrel{\widehat{=}} idS := \emptyset$

$$
\begin{array}{|l}
\underline{Out} \\
\quad \Delta S \\
\quad v! : ID \\
\hline
\quad v! \notin idS \\
\quad idS' = idS \cup \{v!\} \\
\end{array}
$$

$$
\begin{array}{|l}
\underline{Remove} \\
\quad \Delta S \\
\quad x? : ID \\
\hline
\quad idS' = idS \setminus \{x?\} \\
\end{array}
$$

    $\bullet\ Init\, ;$
       var $v : ID \bullet (\mu X \bullet (req \to Out\, ;\ out!v \to Skip \ \square\ ret?x \to Remove)\, ;\ X)$

end

Fig. 1. Simple *Circus* specification: fresh identifier generator

The process FIG specifies the system; it is a basic process defined as a sequence of process paragraphs that specify its state and behaviour. The state is defined using a (horizontal) Z schema; in our example it contains just one component: the set idS of identifiers currently in use.

The behaviour of a process is given by a main action at the end of its specification. In our example, first of all, it uses the action $Init$ to initialise the state: it assigns the empty set to idS. Afterwards, a local variable v is declared, and a recursion is used to define that FIG repeatedly offers to its environment the choice of requesting or returning an identifier. After a request via a synchronisation on req, the action Out, which is specified by a Z schema, is used to define the value of v to be that of any unused identifier, which is then recorded in idS. The value of v is output via the channel out. If, on the other hand, an identifier x is returned via ret, then the action $Remove$ is used to update the state.

As shown in our small example, an action can be defined using a combination of Z, CSP, and imperative programming constructs. It can be a data operation specified in Z, or an assignment for example. It can also be *Skip*, the action that terminates immediately, without interacting with the environment, or *Stop*, the

action that deadlocks. More interestingly, an action can interact with the environment via channels that can be used for input and output of values. Process algebra constructs like parallelism and external choice can be used to combine actions that involve both communications and data operations.

In addition, processes can also be combined using CSP operators. For example, we can combine *FIG* in parallel with another process that uses it to provide identifiers for new employees, for instance. In this case, the channels *req*, *ret*, and *out* are likely to be internal to the system, and can be hidden like in CSP.

Actions are modelled as predicates of a restriction of the UTP theory for CSP, in which the state components and local variables in scope, and their dashed counterparts, are part of the alphabet, in addition to the extra variables *ok*, *wt*, *tr*, and *ref*, and their dashed counterparts. Action refinement is characterised by reverse implication just like in the UTP. Process refinement, on the other hand, is characterised by refinement of the main actions, with the state components taken as local variables. This follows from the fact that the state of a process is encapsulated, and its behaviour is given by the main action.

In the sequel, we calculate a characterisation of traces refinement and *conf* for the UTP theory for CSP, and, therefore, for **Circus** actions and processes. We also establish that, jointly, they are equivalent to process refinement.

4   Traces refinement

In the **Circus**, or CSP, theory of the UTP, the boolean variable *ok* records whether or not a process is in a divergent state (of another process that has already started). If the state is not divergent, that is, if *ok* holds, then *wt*, also a boolean observational variable, determines whether the previous process is waiting for interaction or has terminated. The sequence of events *tr* gives the history of interactions of the previous process, and finally, *ref* gives a set of events in which it may refuse to engage. Similarly, the dashed variables ok', wt', tr' and ref' give similar information about the current process.

A number of healthiness conditions characterise first reactive processes in general, and then those that are in the CSP theory. The **Circus** theory has an extra healthiness condition. Here we use the healthiness conditions **R2** and **CSP4**, which we describe below. A complete discussion can be found in [18].

The healthiness condition **R2** requires that an action does not rely on the history of interactions that passed before its activation, that is, *tr*, and restricts only the new events to be recorded since the last observation, that is, $tr' - tr$. It has two different formulations; we use the one shown below.

$$\textbf{\textit{R2}}(A) \;\; \widehat{=} \;\; A[\langle\rangle, tr' - tr/tr, tr']$$

This requires that the action A is not changed if *tr* is taken to be the empty sequence, and tr' to be just the new events arising from the execution of A.
The condition **CSP4** requires that *Skip* is a right-unit for sequence.

$$\textbf{\textit{CSP4}}(A) \;\; \widehat{=} \;\; A \,;\, Skip$$

In more intuitive terms, **CSP4** requires that, on termination or divergence,

the value of ref' is irrelevant. The following lemma [7] makes this clear; for completeness its proof is presented in the appendix, along with the proof of all other lemmas used in this paper.

Lemma 1.

$$A \mathrel{;} Skip = (\exists \, ref' \bullet A) \land ok' \land \neg \, wt' \ \lor \ A \land ok' \land wt' \ \lor \ (A \land \neg \, ok') \mathrel{;} tr \leq tr'$$

This result shows that, if $A = A \mathrel{;} Skip$, then if A has terminated without diverging, the value of ref' is not relevant. If A has not terminated, then the value of ref' is as defined by A itself. Finally, if it diverges, then the only guarantee is that the trace is extended; the value of the other variables is irrelevant.

We define $A^n \mathrel{\widehat{=}} ok \land \neg \, wt \land A \land ok'$ as the predicate that gives the behaviour of the action A when its preceding action has not diverged and has terminated, and when A itself does not lead to divergence. This is the normal behaviour of A; behaviour in other situations is defined by healthiness conditions. The terminating, non-diverging behaviour of A is $A^t \mathrel{\widehat{=}} A^n \land ok' \land \neg \, wt'$, and finally, the diverging behaviour of A is $A^d \mathrel{\widehat{=}} ok \land \neg \, wt \land A \land \neg \, ok'$. We define that an action A is divergence free if, and only if, $[\neg \, A^d]$.

The function $traces$ defined below [7] gives the set of traces of a *Circus* action defined as a UTP predicate A. This gives a traces model to A compatible with that adopted in the failures-divergences model of CSP.

As already said, the behaviour of the action itself is that prescribed when ok and $\neg \, wt$. The behaviour in the other cases is determined by healthiness conditions of the UTP theory. For example, in the presence of divergence, that is, when $\neg \, ok$, every action can only guarantee that the trace is only extended, so that past history is not modified. This behaviour is not recorded by $traces(A)$.

$$traces(A) = \{ \, tr' - tr \mid A^n \, \} \cup \{ \, (tr' - tr) \mathbin{\frown} \langle \, \checkmark \, \rangle \mid A^t \, \}$$

As mentioned above, tr records the history of interactions before the start of the action; tr' carries this history forward. Therefore, the traces in $traces(A)$ are sequences $tr' - tr$ obtained by removing from tr' its prefix tr. In addition, if $tr' - tr$ leads to termination, then $traces(A)$ also includes $(tr' - tr) \mathbin{\frown} \langle \, \checkmark \, \rangle$, since $\checkmark$ is used in the failures-divergences model to signal termination.

The properties of $traces(A)$ depend on those of A. Since the UTP actions do not satisfy all healthiness conditions imposed on the failures-divergences model, there are sets of traces that do not correspond to any of those of a CSP process. For example, $\boldsymbol{R}(true \vdash tr' = tr \mathbin{\frown} \langle \, a, b \, \rangle \land \neg \, wt')$ is an action that engages in the events a and b and then terminates. Its behaviour does not allow for the traces $\langle \, \rangle$ and $\langle \, a \, \rangle$, so its set of traces does not include the empty trace and is not prefix closed as required in the failures-divergences model.

The divergent behaviour of a UTP action in the theory of CSP processes does not enforce $\neg \, ok'$; the healthiness condition $\boldsymbol{CSP2}$ enforces exactly that, whenever $\neg \, ok'$ is possible, so is ok'. This means that no process is required to diverge, and that one of the possible behaviours of a divergent process is not to diverge or even terminate. For example, the behaviour of the divergent process

Chaos, when *ok* and $\neg$ *wt* hold, is given simply by $tr \leq tr'$. This means that *traces*(*Chaos*), for example, includes every possible trace. This is in contradiction with the traces model of CSP, where the process that diverges immediately is identified with *Stop* in the traces model [19]; its only trace is the empty trace.

In this work, however, since we are interested only in divergence-free actions, this is not an issue. In [7], we have actually introduced the set $traces_\perp(A)$, which is defined as follows to include all traces that lead to divergence.

$$traces_\perp(A) = traces(A) \cup divergences(A)$$
$$divergences(A) = \{\, tr' - tr \mid A^d \,\}$$

For **CSP2** reactive actions A, the sets $traces(A)$ and $traces_\perp(A)$ are the same, because the traces that lead to divergence may also lead to non-divergence, and so are included in $traces(A)$. Since divergence-free actions are **CSP2**, and in any case we are interested in (models of) *Circus* actions and processes, which are **CSP2**, it is adequate for us to use $traces(A)$ in our work. In addition, for divergence-free actions A, the set $traces(A)$ is that in the traces model of CSP, which is also the set $traces_\perp(A)$ defined in the failures-divergences model.

Here, using the connection between the UTP theory and the CSP traces model defined by *traces*, we now calculate a characterisation for traces refinement in the UTP for divergence-free actions. Refinement in the UTP is defined for predicates on the same alphabet. In the case of traces refinement, it is defined for CSP processes, and so, for *Circus* actions in particular, but there is no need to assume that the programming variables in their alphabets are the same. In what follows, we consider actions A_1 and A_2, whose alphabets include the lists v_1 and v_2 of undashed variables. Both v_1 and v_2 include *ok*, *wt*, *tr*, and *ref*, but also possibly different (lists of) variables x_1 and x_2 representing state components.

The proof of the theorem below, and of the others in the sequel, use a few lemmas stated and proved in the appendix; in particular, the next theorem uses Lemma 2. We use $[\, A \,]$ as an abbreviation for a universal quantification over all variables v_1, v_1', v_2, and v_2'. As expected, for a list of variables v, the list v' contains the corresponding dashed variables.

Theorem 1.

$$A_1 \sqsubseteq_T A_2 \Leftrightarrow [A_2^n \Rightarrow (\exists\, w_1, w_1' \bullet A_1^n) \wedge (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t)]$$

where the variable list $v_1 = w_1, tr$, and provided A_1 and A_2 are divergence free.

Proof

$A_1 \sqsubseteq_T A_2$

$\Leftrightarrow traces(A_2) \subseteq traces(A_1)$ [definition of traces refinement]

$\Leftrightarrow \{\, tr' - tr \mid A_2^n \,\} \cup \{\, (tr' - tr) ^\frown \langle \checkmark \rangle \mid A_2^t \,\}$ [definition of *traces*]
     $\subseteq \{\, tr' - tr \mid A_1^n \,\} \cup \{\, (tr' - tr) ^\frown \langle \checkmark \rangle \mid A_1^t \,\}$

$\Leftrightarrow \{\, tr' - tr \mid A_2^n \,\} \subseteq \{\, tr' - tr \mid A_1^n \,\} \land \{\, tr' - tr \mid A_2^t \,\} \subseteq \{\, tr' - tr \mid A_1^t \,\}$

$$\qquad\qquad\qquad [\text{property of sets and } \checkmark \text{ not in the range of } tr \text{ or } tr']$$

$$\Leftrightarrow \left(\begin{array}{l} (\forall\, t \bullet (\exists\, v_2, v_2' \bullet A_2^n \land t = tr' - tr) \Rightarrow (\exists\, v_1, v_1' \bullet A_1^n \land t = tr' - tr)) \land \\ (\forall\, t \bullet (\exists\, v_2, v_2' \bullet A_2^t \land t = tr' - tr) \Rightarrow (\exists\, v_1, v_1' \bullet A_1^t \land t = tr' - tr)) \end{array} \right)$$

$$\qquad\qquad\qquad\qquad\qquad\qquad [\text{property of sets}]$$

$$\Leftrightarrow \left(\forall\, t, v_2, v_2' \mid t = tr' - tr \land A_2^n \bullet \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^n \land t = tr' - tr) \land \\ (\neg\, wt' \Rightarrow \exists\, v_1, v_1' \bullet A_1^t \land t = tr' - tr) \end{array} \right) \right)$$

$$\qquad\qquad\qquad\qquad [\text{predicate calculus, and definitions of } A_2^t \text{ and } A_2^n]$$

$$\Leftrightarrow \left(\forall\, t, v_2, v_2' \mid t = tr' - tr \land A_2^n \bullet \left(\begin{array}{l} (\exists\, w_1, w_1' \bullet A_1^n[\langle\rangle, t/tr, tr']) \land \\ (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t[\langle\rangle, t/tr, tr']) \end{array} \right) \right)$$

$$\qquad\qquad\qquad\qquad\qquad\qquad [\text{Lemma 2}]$$

$$\Leftrightarrow \left(\forall\, v_2, v_2' \bullet A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, w_1, w_1' \bullet A_1^n[\langle\rangle, tr' - tr/tr, tr']) \land \\ (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t[\langle\rangle, tr' - tr/tr, tr']) \end{array} \right) \right)$$

$$\qquad\qquad\qquad\qquad\qquad\qquad [\text{predicate calculus}]$$

$\Leftrightarrow \forall\, v_2, v_2' \bullet A_2^n \Rightarrow (\exists\, w_1, w_1' \bullet A_1^n) \land (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t)$ $\qquad$ [**R2**]

$\Leftrightarrow [A_2^n \Rightarrow (\exists\, w_1, w_1' \bullet A_1^n) \land (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t)]$ $\qquad$ [predicate calculus]

$\square$

In words, this characterisation of traces refinement establishes that if t is a trace of A_2, then it is a trace of A_1, and if it leads to termination for A_2, then it also leads to termination for A_1.

We observe that, if a trace is not terminating for A_2, then it may or may not be terminating for A_1. If it is not terminating for A_2 because A_2 deadlocks, but it is terminating for A_1, we have a situation in which termination is refined by deadlock. Indeed, in the simplest case, we observe that *Skip* is refined by *Stop*; in fact, *Stop* is the most refined CSP process according to the traces refinement relation. If, on the other hand, a trace is not terminating for A_2 because it proceeds to carry out further interactions, but A_1 terminates, for the extension of the trace, the required property for traces refinement does not hold.

5   The *conf* relation

The well-studied satisfaction relation [4] called *conf*, for conformance relation, can be defined in terms of failures. A failure of a process P is a pair (t, X), where t is a trace of P, and X is a set of events in which it may refuse to engage after performing the events in t (in the order determined by t).

The *conf* relation is defined for divergence-free processes. The function defined below gives the set of failures of a divergence-free action A.

$$
\begin{aligned}
failures(A) = \{\, & ((tr' - tr), ref') \mid A^n \,\} \cup \\
& \{\, ((tr' - tr), ref' \cup \{\,\checkmark\,\}) \mid A^n \wedge wt' \,\} \cup \\
& \{\, ((tr' - tr) \frown \langle\,\checkmark\,\rangle, ref') \mid A^t \,\} \cup \\
& \{\, ((tr' - tr) \frown \langle\,\checkmark\,\rangle, ref' \cup \{\,\checkmark\,\}) \mid A^t \,\}
\end{aligned}
$$

In a state that is not terminating, for every refusal set ref', there is an extra set $ref' \cup \{\,\checkmark\,\}$. This is because $\checkmark$ is not part of the UTP model and is not considered in the definition of ref', just as it is not considered in the definition of tr'. As before, for a terminating state, the extra trace $(tr' - tr) \frown \langle\,\checkmark\,\rangle$ is recorded. Finally, after termination, $\checkmark$ is also refused, and so $ref' \cup \{\,\checkmark\,\}$ is included.

For actions A_1 and A_2, *conf* can be defined as follows.

$$
A_2 \ conf \ A_1 \ \widehat{=} \ \forall\, t : traces(A_1) \cap traces(A_2) \bullet Ref(A_2, t) \subseteq Ref(A_1, t)
$$
$$
\text{where } Ref(A, t) \ \widehat{=} \ \{\, X \mid (t, X) \in failures(A) \,\}
$$

The above definition of $Ref(A, t)$ is compatible with the definition of $refusals(P)$ in CSP, for the process P/t [19, pages 94,197]. Intuitively, the action A_2 conforms to another action A_1 if, and only if, whenever A_2 performs a trace of events that is also possible for A_1, it does not refuse more events than A_1. In other words, deadlock is reduced or maintained after common traces.

The following theorem gives a characterisation of *conf* for the UTP. It is a relation between divergence-free actions.

Theorem 2.

$$
A_2 \ conf \ A_1 \Leftrightarrow \left[(\exists\, w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right]
$$

where $v_1 = w_1, tr$, and $w_1 = k_1, ref$, and A_1 and A_2 are divergence free.

Proof

$A_2 \ conf \ A_1$

$\Leftrightarrow \forall\, t : traces(A_1) \cap traces(A_2) \bullet Ref(A_2, t) \subseteq Ref(A_1, t)$ \quad [definition of *conf*]

$$
\Leftrightarrow \left(\left(\begin{array}{l} \forall\, t : traces(A_1) \cap traces(A_2) \bullet \\ \forall\, v_2, v_2' \bullet \left(\begin{array}{l} \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \Rightarrow \\ \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt' \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \Rightarrow \\ \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \end{array} \right) \end{array} \right) \end{array} \right) \right)
$$

[Lemma 3]

$$\Leftrightarrow \forall\, t \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^n \wedge (t = tr' - tr \vee \neg\, wt' \wedge t = (tr' - tr) \frown \langle\checkmark\rangle)) \wedge \\ (\exists\, v_2, v_2' \bullet A_2^n \wedge (t = tr' - tr \vee \neg\, wt' \wedge t = (tr' - tr) \frown \langle\checkmark\rangle)) \end{array} \right) \\ \Rightarrow \\ \left(\forall\, v_2, v_2' \bullet \left(\begin{array}{l} \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt' \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \end{array} \right) \end{array} \right) \right) \end{array} \right) \right)$$

[definition of *traces* and property of sets]

$$\Leftrightarrow \forall\, t \bullet \left(\left(\begin{array}{l} \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^n \wedge t = tr' - tr) \wedge (\exists\, v_2, v_2' \bullet A_2^n \wedge t = tr' - tr) \vee \\ \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle) \wedge \\ (\exists\, v_2, v_2' \bullet A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle) \end{array} \right) \end{array} \right) \\ \Rightarrow \\ \left(\forall\, v_2, v_2' \bullet \left(\begin{array}{l} \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt' \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \end{array} \right) \end{array} \right) \right) \end{array} \right) \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\begin{array}{l} \left(\begin{array}{l} \forall\, t, v_2, v_2' \bullet \\ (\exists\, v_1, v_1' \bullet A_1^n \wedge t = tr' - tr) \\ \Rightarrow \\ \left(\begin{array}{l} (\exists\, v_2, v_2' \bullet A_2^n \wedge t = tr' - tr) \Rightarrow \\ \left(\begin{array}{l} \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt' \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \wedge \\ \left(\begin{array}{l} \forall\, t, v_2, v_2' \bullet \\ (\exists\, v_1, v_1' \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle) \wedge \\ \Rightarrow \\ \left(\begin{array}{l} (\exists\, v_2, v_2' \bullet A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle) \Rightarrow \\ \left(\begin{array}{l} \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt' \end{array} \right) \wedge \\ \left(\begin{array}{l} A_2^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \Rightarrow \\ \quad \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) \frown \langle\checkmark\rangle \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\left\{ \begin{array}{l} \forall\, t, v_2, v_2' \bullet \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^n \wedge t = tr' - tr) \wedge A_2^n \wedge t = tr' - tr \\ \Rightarrow \\ \left(\begin{array}{l} (\exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr) \wedge \\ (wt' \Rightarrow \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt') \end{array} \right) \end{array} \right) \wedge \\ \forall\, t, v_2, v_2' \bullet \left(\begin{array}{l} \left(\begin{array}{l} (\exists\, v_1, v_1' \bullet A_1^n \wedge \neg\, wt' \wedge t = (tr' - tr) \frown \langle \checkmark \rangle) \wedge \\ A_2^n \wedge \neg\, wt' \wedge t = (tr' - tr) \frown \langle \checkmark \rangle \end{array} \right) \\ \Rightarrow \\ (\exists\, u_1, u_1', ref \bullet A_1^n \wedge \neg\, wt' \wedge t = (tr' - tr) \frown \langle \checkmark \rangle) \end{array} \right) \end{array} \right\} \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\begin{array}{l} \left(\forall\, v_2, v_2' \bullet (\exists\, w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \wedge \\ (\forall\, v_2, v_2' \bullet (\exists\, w_1, w_1' \bullet A_1^t) \wedge A_2^t \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^t) \end{array} \right)$$

[Lemma 2, **R2**, $v_1 = w_1, tr$ and $v_1 = k_1, tr, ref$]

$$\Leftrightarrow \left(\begin{array}{l} \left(\forall\, v_2, v_2' \bullet (\exists\, w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \wedge \\ (\forall\, v_2, v_2' \bullet (\exists\, k_1, k_1', ref \bullet A_1^t) \wedge A_2^t \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^t) \end{array} \right)$$

[Lemma 4]

$$\Leftrightarrow \left[(\exists\, w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow (\exists\, k_1, k_1', ref \bullet A_1^n \wedge wt')) \end{array} \right) \right]$$

[predicate calculus]

$\square$

This establishes that, if a trace of A_2 is also a trace of A_1, with any refusal, then (1) it must be possible for A_1 to have that trace with the same refusal; and (2) if the trace leads to an intermediate state of A_2, then it should also lead to an intermediate state of A_1 (with the same refusal). If it leads to a terminating state of A_2, then A_1 may or may not terminate, but must have the same refusals. This stresses the fact that *Skip conf Stop*, but not *Stop conf Skip*, a fact that is perhaps not so obvious in the original definition.

6   Process refinement

Refinement of *Circus* processes is defined as shown below, where we consider two processes P_1 and P_2 whose (lists of) states components are x_1 and x_2, and whose main actions are A_1 and A_2; for simplicity, we omit types.

$$P_1 \sqsubseteq_P P_2 \;\widehat{=}\; (\mathbf{var}\ x_1 \bullet A_1) \sqsubseteq (\mathbf{var}\, x_2 \bullet A_2)$$

The variable blocks make the state components local to the actions. Precisely, the UTP model of a *Circus* variable block is defined as follows.

$$(\mathbf{var}\ x \bullet A) \;\widehat{=}\; (\exists\, x, x' \bullet A)$$

In the definition of process refinement, the alphabets of the actions $(\mathbf{var}\ x_1 \bullet A_1)$

and (**var** x_2 • A_2) are the same; it includes no programming variables. The refinement relation between actions is the standard UTP relation.

Below, we establish that process refinement can be characterised in terms of traces refinement and *conf*. This establishes that we can determine refinement just by examining the traces and refusals of a process. We do not need information about its internal state, to which an observer has no access.

As already mentioned, in our previous work, we have established that traces refinement and *conf* correspond to failures-divergences refinement in CSP. Here, we show that they do not establish refinement in the richer model of *Circus* processes in the UTP. Instead, it corresponds to processes refinement; this clarifies the role of data in testing for traces inclusion and deadlock reduction.

Theorem 3. *Provided P_1 and P_2 are divergence-free Circus processes with main actions A_1 and A_2, we can characterise refinement as follows.*

$$P_1 \sqsubseteq_P P_2 \Leftrightarrow A_1 \sqsubseteq_T A_2 \wedge A_2 \ conf \ A_1$$

Proof

$A_1 \sqsubseteq_T A_2 \wedge A_2 \ conf \ A_1$

$$\Leftrightarrow \left(\begin{array}{l} [A_2^n \Rightarrow (\exists w_1, w_1' \bullet A_1^n) \wedge (\neg \ wt' \Rightarrow \exists w_1, w_1' \bullet A_1^t)] \wedge \\ \left[(\exists w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right] \end{array} \right)$$

[Theorems 1 and 2]

$$\Leftrightarrow \left[\begin{array}{l} (A_2^n \Rightarrow (\exists w_1, w_1' \bullet A_1^n) \wedge (\neg \ wt' \Rightarrow \exists w_1, w_1' \bullet A_1^t)) \wedge \\ \left((\exists w_1, w_1' \bullet A_1^n) \wedge A_2^n \Rightarrow \left(\begin{array}{l} (\exists k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right]$$

[predicate calculus]

$$\Leftrightarrow \left[\begin{array}{l} (A_2^n \Rightarrow (\exists w_1, w_1' \bullet A_1^n) \wedge (\neg \ wt' \Rightarrow \exists w_1, w_1' \bullet A_1^t)) \wedge \\ \left(\begin{array}{l} \left((\exists w_1, w_1' \bullet A_1^n) \Rightarrow \left(\begin{array}{l} (\exists k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \vee \\ (A_2^n \Rightarrow ((\exists k_1, k_1', ref \bullet A_1^n) \wedge (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt'))) \end{array} \right) \end{array} \right]$$

[predicate calculus]

$$\Leftrightarrow \left[\begin{array}{l} \left(\begin{array}{l} (A_2^n \Rightarrow (\exists w_1, w_1' \bullet A_1^n) \wedge (\neg \ wt' \Rightarrow \exists w_1, w_1' \bullet A_1^t)) \wedge \\ \left((\exists w_1, w_1' \bullet A_1^n) \Rightarrow \left(\begin{array}{l} (\exists k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right) \vee \\ \left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg \ wt' \Rightarrow \exists w_1, w_1' \bullet A_1^t) \wedge \\ (wt' \Rightarrow \exists k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right]$$

[predicate calculus]

$$\Leftrightarrow \left[\begin{array}{l} \left(\begin{array}{l} (A_2^n \Rightarrow \exists\, w_1, w_1' \bullet A_1^n) \wedge (A_2^n \Rightarrow (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t)) \wedge \\ \left((\exists\, w_1, w_1' \bullet A_1^n) \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right) \vee \\ \left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right]$$

$$\text{[predicate calculus]}$$

$$\Leftrightarrow \left[\begin{array}{l} \left(\begin{array}{l} \left((A_2^n \vee (\exists\, w_1, w_1' \bullet A_1^n)) \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \wedge \\ (A_2^n \Rightarrow (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t)) \end{array} \right) \vee \\ \left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t) \wedge \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \end{array} \right]$$

$$\text{[predicate calculus]}$$

$$\Leftrightarrow \left[\left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, w_1, w_1' \bullet A_1^t) \\ (wt' \Rightarrow \exists\, k_1, k_1', ref \bullet A_1^n \wedge wt') \end{array} \right) \right) \right] \qquad \text{[predicate calculus]}$$

$$\Leftrightarrow \left[\left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, w_1, ok', ref', x' \bullet A_1^n[false/wt']) \\ (wt' \Rightarrow \exists\, k_1, ok', x' \bullet A_1^n[true/wt']) \end{array} \right) \right) \right] \quad \text{[one-point rule]}$$

$$\Leftrightarrow \left[\left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, w_1, ok', ref', x' \bullet A_1^n) \\ (wt' \Rightarrow \exists\, k_1, ok', x' \bullet A_1^n) \end{array} \right) \right) \right] \qquad \text{[predicate calculus]}$$

$$\Leftrightarrow \left[\left(A_2^n \Rightarrow \left(\begin{array}{l} (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge \\ (\neg\, wt' \Rightarrow \exists\, x_1, x_1' \bullet A_1^n) \\ (wt' \Rightarrow \exists\, x_1, x_1' \bullet A_1^n) \end{array} \right) \right) \right]$$

$$[ok,\ wt,\ ref,\ ok',\ \text{and}\ ref'\ \text{are not free in}\ \neg\, wt'\ \text{and}\ wt']$$

$$\Leftrightarrow [A_2^n \Rightarrow (\exists\, k_1, k_1', ref \bullet A_1^n) \wedge (\exists\, x_1, x_1' \bullet A_1^n)] \qquad\qquad \text{[predicate calculus]}$$

$$\Leftrightarrow [A_2^n \Rightarrow (\exists\, x_1, x_1' \bullet A_1^n)] \qquad\qquad\qquad\qquad \text{[predicate calculus]}$$

$$\Leftrightarrow [(\exists\, x_2, x_2' \bullet A_2^n) \Rightarrow (\exists\, x_1, x_1' \bullet A_1^n)] \qquad\qquad [x_2\ \text{and}\ x_1\ \text{are not free in}\ A_1]$$

$$\Leftrightarrow P_1 \sqsubseteq_P P_2 \qquad\qquad\qquad\qquad \text{[definition of process refinement]}$$

$\square$

Actions do not represent systems in *Circus*. Their effects on state are visible, and refinement is only defined for actions on the same state. Therefore, an account of system testing based on *Circus* needs to rely on process refinement. This does not mean, however, that data does not play a part in a testing technique based on *Circus*; we further discuss this issue in the next section.

7   Conclusions

In this paper we have established the foundation of a testing theory for *Circus*, by calculating definitions for traces refinement and *conf* and proving that, together, they characterise process refinement. We are now in a position to consider how the standard techniques of test generation to establish traces refinement and conformance can be applied to the state-rich operational semantics of *Circus*.

Formalisation of testing techniques in the UTP has also been considered in [1]. That work is concerned with fault-based testing using mutations; it goes well beyond what we present here, in that it already provides test-case generation techniques. It is not, however, concerned with testing for refinement in state-rich reactive languages. The formalisation is conducted in the theory of designs for total correctness of sequential imperative programs.

A predicative account of traces refinement is also presented in [9]. In that work, traces refinement is defined for abstract data types, and characterised using simulation relations. It is also observed that *conf* cannot be treated in the same way, because it is not a preorder.

We have already defined exhaustive test sets for CSP processes in [5]. For *Circus*, the operational semantics is defined symbolically, with events referring to values that are constrained by the state and local variable definitions and by the data operations. It supports the integration of model checking and theorem proving techniques in reasoning about *Circus* processes and actions. For testing, the symbolic operational semantics provides guidance for the coverage of traces (by giving structure to the set of traces of an action) and, therefore, for the construction of tests to establish both traces refinement and *conf*.

In the case of CSP, values are part of event names, and are treated indistinctively. For example, $c.0$, $c.1$, and so on, are just event names. In the case of *Circus*, symbolic traces like $\langle\, c.w_0, d.w_1\,\rangle$, for instance, represent collections of traces; this example, in particular, defines a family of traces that record a communication over a channel c followed by a communication over a channel d, of values w_0 and w_1. The symbolic representation and the constraints that w_0 and w_1 are required to satisfy give us an indication of how to produce test data to achieve acceptable coverage of the collection of traces.

These constraints are raised by the local state of the processes, and by its data operations. Therefore, even though testing for process refinement does not require observation of internal state, the valid traces reflect restrictions that arise from the state operations. It is in our immediate plans to adapt to *Circus* the test generation strategy based on a combination of IOLTS (Input-Output Labelled Transition Systems) and algebraic specifications provided in [16, 15]. We will formalise the proposed technique based on the results presented here.

Traces refinement and *conf* also have value as tools for reasoning about safety and liveness properties of actions; we are yet to explored this aspect of the UTP theory. For traces refinement, further work on healthiness conditions are necessary to allow a closer correspondence with the CSP traces model. Algebraic laws of traces refinement and *conf* is also an interesting topic for future work.

Acknowledgments

References

1. B. Aichernig and He Jifeng. Mutation testing in UTP. *Formal Aspects of Computing*, 2008.
2. G. Bernot, M.-C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387 – 405, 1991.
3. L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343 – 360, 1986.
4. E. Brinksma. A theory for the derivation of tests. In *Protocol Specification, testing and Verification VIII*, pages 63 – 74. North-Holland, 1988.
5. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
6. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for **Circus**. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
7. A. L. C. Cavalcanti and J. C. P. Woodcock. A Tutorial Introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220 – 268. Springer-Verlag, 2006.
8. T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, SE-4(3):178 – 187, 1978.
9. J. Derrick and E. Boiten. More Relational Concurrent Refinement: Traces and Partial Relations. In *REFINE Workshop*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.
10. J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An Experiment in Automatic Generation of Conformance Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29:123 – 146, 1997.
11. J. Gannon, P. McMullin, and R. Hamlet. Data abstraction implementation, specification and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.
12. M.-C. Gaudel. Testing can be formal, too. In *International Joint Conference, Theory And Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82 – 96. Springer-Verlag, 1995.
13. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090 – 1126, 1996.
15. G. Lestiennes. *Contributions au test de logiciel basé sur des spécifications formelles*. PhD thesis, Université de Paris-Sud, 2005.
16. G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs, and datatypes. In *IEEE International Symposium on Software Reliability Engineering*, pages 3 – 14, 2002.

17. C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.
18. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing, online first*, 2007.
19. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
20. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.

A   Some lemmas

Lemma 1.

$$A \,;Skip = (\exists\, ref' \bullet A) \wedge ok' \wedge \neg\, wt' \;\vee\; A \wedge ok' \wedge wt' \;\vee\; (A \wedge \neg\, ok')\,; tr \le tr'$$

Proof We take v to be a list of the undashed variables in the alphabet of A, including ok, wt, tr, ref, and programming variables x.

$A \,;Skip$

$$= \exists\, v_0 \bullet A[v_0/v'] \wedge \boldsymbol{R}(true \vdash tr' = tr \wedge \neg\, wt' \wedge x' = x)[v_0/v]$$
$$\text{[definition of sequence and } Skip \text{ (as a reactive design [18])]}$$

$$= \left(\exists\, v_0 \bullet \left(\begin{pmatrix} A[v_0/v'] \wedge \\ ((wt_0 \wedge ((\neg\, ok_0 \wedge tr_0 \le tr') \vee \amalg[v_0/v])) \vee \\ (\neg\, wt_0 \wedge (ok_0 \Rightarrow ok' \wedge tr' = tr_0 \wedge \neg\, wt' \wedge x' = x_0) \wedge tr_0 \le tr') \end{pmatrix} \right) \right)$$
$$\text{[definition of } \boldsymbol{R} \text{ and property of substitution]}$$

$$= \left(\exists\, v_0 \bullet \left(\begin{matrix} A[v_0/v'] \wedge wt_0 \wedge \neg\, ok_0 \wedge tr_0 \le tr' \vee \\ A[v_0/v'] \wedge wt_0 \wedge ok' \wedge \amalg[v_0/v] \vee \\ A[v_0/v'] \wedge \neg\, wt_0 \wedge \neg\, ok_0 \wedge tr_0 \le tr' \vee \\ A[v_0/v'] \wedge \neg\, wt_0 \wedge ok_0 \wedge ok' \wedge tr' = tr_0 \wedge \neg\, wt' \wedge x' = x_0 \end{matrix} \right) \right)$$
$$\text{[predicate calculus]}$$

$$= \left(\exists\, v_0 \bullet \left(\begin{matrix} A[v_0/v'] \wedge \neg\, ok_0 \wedge tr_0 \le tr' \vee \\ A[v_0/v'] \wedge wt_0 \wedge ok_0 \wedge ok' \wedge \amalg[v_0/v] \vee \\ A[v_0/v'] \wedge \neg\, wt_0 \wedge ok_0 \wedge ok' \wedge tr' = tr_0 \wedge \neg\, wt' \wedge x' = x_0 \end{matrix} \right) \right)$$
$$\text{[predicate calculus]}$$

$$= \left(\begin{matrix} (\exists\, v_0 \bullet A[v_0/v'] \wedge \neg\, ok_0 \wedge tr_0 \le tr') \vee \\ (\exists\, v_0 \bullet A[v_0/v'] \wedge wt_0 \wedge ok_0 \wedge \amalg[v_0/v]) \vee \\ (\exists\, v_0 \bullet A[v_0/v'] \wedge \neg\, wt_0 \wedge ok_0 \wedge ok' \wedge tr' = tr_0 \wedge \neg\, wt' \wedge x' = x_0) \end{matrix} \right)$$
$$\text{[predicate calculus]}$$

$$= \left(\begin{matrix} (A \wedge \neg\, ok');\ (tr \le tr') \vee \\ A \wedge wt' \wedge ok' \vee \\ (\exists\, ref' \bullet A) \wedge \neg\, wt' \wedge ok') \end{matrix} \right) \qquad \text{[definition of sequence and one-point rule]}$$

□

Lemma 2.

$$(\exists\, tr, tr' \bullet A \wedge t = tr' - tr) = A[\langle\,\rangle, t/tr, tr']$$

*provided A is **R2**-healthy.*

Proof

$\exists\, tr, tr' \bullet A \wedge t = tr' - tr$

$\quad = \exists\, tr_1, tr'_1 \bullet A[tr_1, tr'_1/tr, tr'] \wedge t = tr'_1 - tr_1$ [predicate calculus]

$\quad = \exists\, tr_1, tr'_1 \bullet A[\langle\,\rangle, tr' - tr/tr, tr'][tr_1, tr'_1/tr, tr'] \wedge t = tr'_1 - tr_1$ [**R2**]

$\quad = \exists\, tr_1, tr'_1 \bullet A[\langle\,\rangle, tr'_1 - tr_1/tr, tr'] \wedge t = tr'_1 - tr_1$ [property of substitution]

$\quad = \exists\, tr_1, tr'_1 \bullet A[\langle\,\rangle, t/tr, tr'] \wedge t = tr'_1 - tr_1$ [property of equality]

$\quad = A[\langle\,\rangle, t/tr, tr'] \wedge \exists\, tr_1, tr'_1 \bullet t = tr'_1 - tr_1$ [predicate calculus]

$\quad = A[\langle\,\rangle, t/tr, tr']$ [property of sequences]

$\square$

Lemma 3.

$$Ref(A_2, t) \subseteq Ref(A_1, t) \Leftrightarrow$$
$$\begin{bmatrix} (A_2^n \wedge t = tr' - tr \Rightarrow \exists\, u_1, u'_1, ref \bullet A_1^n \wedge t = tr' - tr) \wedge \\ (A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \exists\, u_1, u'_1, ref \bullet A_1^n \wedge t = tr' - tr \wedge wt') \wedge \\ (A_2^t \wedge t = (tr' - tr)\frown\langle\checkmark\rangle \Rightarrow \exists\, u_1, u'_1, ref \bullet A_1^t \wedge t = (tr' - tr)\frown\langle\checkmark\rangle) \end{bmatrix}$$

Proof

$Ref(A_2, t) \subseteq Ref(A_1, t)$

$\Leftrightarrow \{X \mid (t, X) \in failures(A_2)\} \subseteq \{X \mid (t, X) \in failures(A_1)\}$

         [definition of $Ref(A, t)$]

$$\Leftrightarrow \left(\begin{Bmatrix} X \mid (t, X) \in \begin{pmatrix} \{\,((tr' - tr), ref') \mid A_2^n\,\} \cup \\ \{\,((tr' - tr), ref' \cup \{\checkmark\}) \mid A_2^n \wedge wt'\,\} \cup \\ \{\,((tr' - tr)\frown\langle\checkmark\rangle, ref') \mid A_2^t\,\} \cup \\ \{\,((tr' - tr)\frown\langle\checkmark\rangle, ref' \cup \{\checkmark\}) \mid A_2^t\,\} \end{pmatrix} \end{Bmatrix} \right.$$
$$\left. \subseteq \begin{Bmatrix} X \mid (t, X) \in \begin{pmatrix} \{\,((tr' - tr), ref') \mid A_1^n\,\} \cup \\ \{\,((tr' - tr), ref' \cup \{\checkmark\}) \mid A_1^n \wedge wt'\,\} \cup \\ \{\,((tr' - tr)\frown\langle\checkmark\rangle, ref') \mid A_1^t\,\} \cup \\ \{\,((tr' - tr)\frown\langle\checkmark\rangle, ref' \cup \{\checkmark\}) \mid A_1^t\,\} \end{pmatrix} \end{Bmatrix} \right)$$

         [definition of *failures* and property of sets]

$$\Leftrightarrow \left(\forall X \bullet \left(\begin{matrix} \left(\exists\, v_2, v_2' \bullet \left(\begin{pmatrix} A_2^n\ \wedge \\ t = tr' - tr \wedge X = ref' \vee \\ t = tr' - tr \wedge wt' \wedge X = ref' \cup \{\checkmark\} \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \cup \{\checkmark\} \end{pmatrix} \right) \right) \\ \Rightarrow \\ \left(\exists\, v_1, v_1' \bullet \left(\begin{pmatrix} A_1^n\ \wedge \\ t = tr' - tr \wedge X = ref' \vee \\ t = tr' - tr \wedge wt' \wedge X = ref' \cup \{\checkmark\} \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \cup \{\checkmark\} \end{pmatrix} \right) \right) \end{matrix} \right) \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\forall X, v_2, v_2' \bullet \left(\begin{matrix} \left(A_2^n \wedge \begin{pmatrix} t = tr' - tr \wedge X = ref' \vee \\ t = tr' - tr \wedge wt' \wedge X = ref' \cup \{\checkmark\} \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \cup \{\checkmark\} \end{pmatrix} \right) \\ \Rightarrow \\ \left(\exists\, v_1, v_1' \bullet \left(\begin{pmatrix} A_1^n\ \wedge \\ t = tr' - tr \wedge X = ref' \vee \\ t = tr' - tr \wedge wt' \wedge X = ref' \cup \{\checkmark\} \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge X = ref' \cup \{\checkmark\} \end{pmatrix} \right) \right) \end{matrix} \right) \right)$$

[predicate calculus]

$$\Leftrightarrow \left(\forall\, v_2, v_2' \bullet \left(\begin{matrix} \begin{pmatrix} A_2^n \wedge t = tr' - tr \Rightarrow \\ \exists\, u_1, u_1', ref, refX \bullet \\ \begin{pmatrix} A_1^n[refX/ref'] \wedge \\ \begin{pmatrix} t = tr' - tr \wedge ref' = refX \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge ref' = refX \end{pmatrix} \end{pmatrix} \end{pmatrix} \wedge \\ \begin{pmatrix} A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\ \exists\, u_1, u_1', ref, refX \bullet \\ \begin{pmatrix} A_1^n[refX/ref'] \wedge \\ \begin{pmatrix} t = tr' - tr \wedge wt' \wedge ref' = refX \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge ref' = refX \end{pmatrix} \end{pmatrix} \end{pmatrix} \wedge \\ \begin{pmatrix} A_2^n \wedge \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \Rightarrow \\ \exists\, u_1, u_1', ref, refX \bullet \\ \begin{pmatrix} A_1^n[refX/ref'] \wedge \\ \begin{pmatrix} t = tr' - tr \wedge ref' = refX \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge ref' = refX \end{pmatrix} \end{pmatrix} \end{pmatrix} \wedge \\ \begin{pmatrix} A_2^n \wedge \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \Rightarrow \\ \exists\, u_1, u_1', ref, refX \bullet \\ \begin{pmatrix} A_1^n[refX/ref'] \wedge \\ \begin{pmatrix} t = tr' - tr \wedge wt' \wedge ref' = refX \vee \\ \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle\checkmark\rangle \wedge ref' = refX \end{pmatrix} \end{pmatrix} \end{pmatrix} \end{matrix} \right) \right)$$

[predicate calculus, $v_1 = u_1, ref$, and $\checkmark$ is not ref']

$$
\Leftrightarrow
\begin{pmatrix}
\forall\, v_2, v_2' \bullet \\
\left(
\begin{pmatrix}
\begin{pmatrix}
A_2^n \wedge t = tr' - tr \Rightarrow \\
\begin{pmatrix}
\exists\, u_1, u_1', ref, refX \bullet \\
\begin{pmatrix}
A_1^n[refX/ref'] \wedge \\
\begin{pmatrix}
t = tr' - tr \wedge ref' = refX \vee \\
\neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \wedge ref' = refX
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix} \wedge \\
\begin{pmatrix}
A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\
\begin{pmatrix}
\exists\, u_1, u_1', ref, refX \bullet \\
\begin{pmatrix}
A_1^n[refX/ref'] \wedge \\
\begin{pmatrix}
t = tr' - tr \wedge wt' \wedge ref' = refX \vee \\
\neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \wedge ref' = refX
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix} \wedge \\
\begin{pmatrix}
A_2^n \wedge \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \Rightarrow \\
\begin{pmatrix}
\exists\, u_1, u_1', ref, refX \bullet \\
\begin{pmatrix}
A_1^n[refX/ref'] \wedge \\
\begin{pmatrix}
t = tr' - tr \wedge wt' \wedge ref' = refX \vee \\
\neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \wedge ref' = refX
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
\right)
\end{pmatrix}
$$

[predicate calculus]

$$
\Leftrightarrow
\begin{pmatrix}
\forall\, v_2, v_2' \bullet
\begin{pmatrix}
(A_2^n \wedge t = tr' - tr \Rightarrow \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr) \wedge \\
\begin{pmatrix}
A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \\
\quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt'
\end{pmatrix} \wedge \\
\begin{pmatrix}
A_2^n \wedge \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \Rightarrow \\
\quad \exists\, u_1, u_1', ref \bullet A_1^n \wedge \neg\, wt' \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle
\end{pmatrix}
\end{pmatrix}
\end{pmatrix}
$$

[predicate calculus]

$$
\Leftrightarrow
\begin{bmatrix}
(A_2^n \wedge t = tr' - tr \Rightarrow \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr) \wedge \\
(A_2^n \wedge t = tr' - tr \wedge wt' \Rightarrow \exists\, u_1, u_1', ref \bullet A_1^n \wedge t = tr' - tr \wedge wt') \wedge \\
(A_2^t \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle \Rightarrow \exists\, u_1, u_1', ref \bullet A_1^t \wedge t = (tr' - tr) ^\frown \langle \checkmark \rangle)
\end{bmatrix}
$$

[definition of A_2^t and A_1^t]

$\square$

Lemma 4.

$$(\exists\, ref' \bullet A^t) = A^t$$

provided A is **CSP4**-healthy.

Proof

$(\exists\, ref' \bullet A^t)$

$= ok \wedge \neg\, wt \wedge (\exists\, ref' \bullet A) \wedge ok' \wedge \neg\, wt'$ 　　　　　　　[definition of A^t]

$= ok \wedge \neg\, wt \wedge (\exists\, ref' \bullet ((\exists\, ref' \bullet A) \vee ((A \wedge \neg\, ok');\ tr \le tr'))) \wedge ok' \wedge \neg\, wt'$

[Lemma 1]

$= ok \wedge \neg\, wt \wedge ((\exists\, ref' \bullet A) \vee ((A \wedge \neg\, ok');\ tr \le tr')) \wedge ok' \wedge \neg\, wt'$

[ref' is not free in $(\exists\, ref' \bullet A) \vee ((A \wedge \neg\, ok');\ tr \le tr')$]

$$= ok \wedge \neg\, wt \wedge A \wedge ok' \wedge \neg\, wt' \qquad\qquad\qquad [\text{Lemma 1}]$$

$$= A^t \qquad\qquad\qquad\qquad\qquad\qquad [\text{definition of } A^t]$$

$\square$