Information and Software Technology 000 (2016) 1–23

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

# Formal mutation testing for *Circus*

Alex Alberto[a,*], Ana Cavalcanti[b], Marie-Claude Gaudel[c], Adenilso Simão[a]

[a] Universidade de São Paulo, ICMC, São Carlos, Brazil
[b] Department of Computer Science, University of York, York YO10 5GH, UK
[c] LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay, Orsay 91405, France

ARTICLE INFO

*Article history:*
Received 24 August 2015
Revised 22 March 2016
Accepted 4 April 2016
Available online xxx

*Keywords:*
*Circus*
Mutation
Testing
Formal specification

ABSTRACT

**Context:** The demand from industry for more dependable and scalable test-development mechanisms has fostered the use of formal models to guide the generation of tests. Despite many advancements having been obtained with state-based models, such as Finite State Machines (FSMs) and Input/Output Transition Systems (IOTSs), more advanced formalisms are required to specify large, state-rich, concurrent systems. *Circus*, a state-rich process algebra combining Z, CSP and a refinement calculus, is suitable for this; however, deriving tests from such models is accordingly more challenging. Recently, a testing theory has been stated for *Circus*, allowing the verification of process refinement based on exhaustive test sets.

**Objective:** We investigate fault-based testing for refinement from *Circus* specifications using mutation. We seek the benefits of such techniques in test-set quality assertion and fault-based test-case selection. We target results relevant not only for *Circus*, but to any process algebra for refinement that combines CSP with a data language.

**Method:** We present a formal definition for fault-based test sets, extending the *Circus* testing theory, and an extensive study of mutation operators for *Circus*. Using these results, we propose an approach to generate tests to kill mutants. Finally, we explain how prototype tool support can be obtained with the implementation of a mutant generator, a translator from *Circus* to CSP, and a refinement checker for CSP, and with a more sophisticated chain of tools that support the use of symbolic tests.

**Results:** We formally characterise mutation testing for *Circus*, defining the exhaustive test sets that can kill a given mutant. We also provide a technique to select tests from these sets based on specification traces of the mutants. Finally, we present mutation operators that consider faults related to both reactive and data manipulation behaviour. Altogether, we define a new fault-based test-generation technique for *Circus*.

**Conclusion:** We conclude that mutation testing for *Circus* can truly aid making test generation from state-rich model more tractable, by focussing on particular faults.



## 1. Introduction

Testing from formal models is currently advancing as a solid approach to support the growing demand from industry for more dependable and scalable test-development mechanisms. For instance, Model-Based Testing (MBT) benefits greatly from a precise and clear semantics for models, as opposed to informal or semi-formal models whose semantics is dependent on the particular tool in use.

Many advancements have been obtained with state-based models, such as Finite State Machines (FSMs) [1–10] and Input/Output Transition Systems (IOTSs) [11–15]. Those models, however, quickly become intractable when dealing with larger systems. Thus, more advanced formalisms are required to facilitate the specification of large, state-rich, concurrent systems.

*Circus* is a state-rich process algebra combining Z [16], CSP [17], and a refinement calculus [18]. Its denotational and operational semantics are based on the Unifying Theories of Programming (UTP) [19]. *Circus* can be used to verify large concurrent systems, including those that cannot be handled by model checking. *Circus* has already been used to verify, for example, software in aerospace applications [20], and novel virtualization software by the US Naval Research Laboratory [21].

A theory of testing for *Circus* [22], instantiating Gaudel's long-standing theory of formal testing [23–25], is available. It is founded

* Corresponding author. Tel.: +5516991582600.
*E-mail address:* alexdba@gmail.com (A. Alberto).

http://dx.doi.org/10.1016/j.infsof.2016.04.003
0950-5849/

on the *Circus* operational semantics [26], described and justified in the UTP [27]. As usual in testing, it considers divergence-free processes for the model and the system under test. More precisely, if a system under test diverges, since one cannot decide whether it is deadlocked or divergent, divergence is assimilated to an unspecified deadlock and detected as a failure.

The *Circus* testing theory introduces potentially infinite (symbolic) exhaustive test sets. To achieve practical usefulness, it is, therefore, mandatory to rely on selection criteria both to generate and to select a finite set of tests.

Test-case generation in model-based testing is guided by testing requirements that should be met by a test suite. Usually, the requirements are either coverage criteria that state which elements of the model should be traversed (covered) by test execution, or fault models, which define specific faults that the test cases are supposed to reveal, if present in the system. These approaches are usually complementary to each other. Coverage-based testing is proposed for *Circus* in [28]. It is worth investigating how fault-based testing can complement the coverage testing.

Mutation testing is recognised as one of the most effective fault-detection techniques [29]. The systematic injection of feasible modelling faults into specifications allows the prediction of potential defective implementations. The faults are seeded by syntactic changes that may affect the observable specified behaviour. Such faulty models are "mutants". A mutant is "killed" by a test case able to expose its observable behaviour difference. Testing can benefit from mutation in two ways [30]: some quality aspects of a test set can be measured by the number of mutants it can kill, and the analysis of a mutant model allows the selection of tests targeting specific faults or fault classes.

In this paper, we introduce an approach to apply mutation testing to *Circus* specifications. Most of the presented mutation operators, that is, the fault-injection strategies, are based on previous works that have tackled similar challenges in related modelling languages [31–33]. The outcome of all mutation operators are, however, analyzed considering the specific features and particularities of *Circus*. Moreover, our results are valid in the context of other process algebras, especially those based on CSP [34,35].

The contribution of this paper is manifold. First, we instantiate the notions of mutation testing for a state-rich concurrent language, namely, *Circus*, and its formal theory of testing. In particular, we face the challenge of associating mutations in the text of a *Circus* specification to traces of the *Circus* denotational semantics that define tests that cover the mutation. Even though mutation testing has already been applied to languages and theories upon which *Circus* is based, such as CSP [31] and the UTP [36], the consideration of a state-rich process algebra for refinement with a UTP semantics is novel. Second, we propose mutation operators for *Circus*, analysing and adapting existing ones for the underlying languages and designing some that are specific to *Circus*. Third, we describe prototype tool support for the application of the mutant operators and two approaches to generate tests that can kill these mutants. When it is feasible to translate the considered *Circus* specification into CSP, we propose the use of the FDR model checker. For the other cases, we identify a tool chain that copes directly with *Circus* specifications via slicing techniques and symbolic execution.

This paper is organized as follows. Section 2 gives an overview of the aspects of *Circus* and its testing theory that we use here. Section 3 extends the testing theory to consider mutation testing and describes our approach to generating tests based on mutants. The mutation operators used to generate the mutants themselves are defined in Section 4. Tool support for automation of our approach is discussed in Section 5, and an extra complete example is introduced in Section 6. Finally, we present some related and future work and conclusions in Section 7 and 8.

## 2. *Circus* and its testing theory

In this section, we give a brief description of the *Circus* language, its operational semantics [26], and its testing theory [22].

### 2.1. *Circus* notation and operational semantics

As exemplified is Fig. 1, *Circus* allows us to model systems and their components via (a network of) interacting processes. In Fig. 1, we define a single process *Chrono* that specifies the reactive behaviour of a chronometer. This is a process that recognises *tick* events that mark the passage of time, a request to output the current time via a channel *time*, and outputs minutes and seconds via a channel *out*.

A *Circus* specification is defined by a sequence of paragraphs. Roughly speaking, they define processes, but also channels, and any types and functions used in the process specifications. In Fig. 1, we define a type *RANGE*, including the valid values for seconds and minutes, and the channels *tick, time* and *out*. The channel *out* is typed, since it is used to communicate the current minutes and seconds recorded in the chronometer as a pair. The final paragraph in Fig. 1 defines *Chrono* itself.

Each process has:

- **a state** and some operations for observing and changing it in a Z style. In *Chrono*, the state is composed by a pair *AState* of variables named sec and min with integer values between 0 and 59 (as defined by *RANGE*), and the data operations on this state are specified by the three schemas *AInit, IncSec, IncMin*.
- **actions** that define communicating behaviours in a CSP style. The overall behaviour of a process is specified by the main action after the symbol $\cdot$. In our example, it is a sequential composition of the schema *AInit* followed by the repeated execution of the *Run* action. The *Circus* construct $\mu X \cdot A(X)$ defines a recursive action $A$, in which $X$ is used for recursive calls.

The initialisation schema *AInit* defines the values sec' and min' of the state components after the initialisation. These components are declared using $AState'$. The operation schemas *IncSec* and *IncMin* change the state, as indicated by the declaration $\Delta AState$. They also define values sec' and min' of the state components after the operations. In each case, the seconds and minutes are incremented modulo 60.

*Run* starts with an external choice ($\Box$) between the events *tick* and *time*. If the environment chooses the event *tick*, this is followed by the increment of the chronometer using the data operation *IncSec*. Afterwards, we have another choice between actions guarded by the conditions $sec = 0$ and $sec \neq 0$. If, after the increment, we have $sec = 0$, then the minutes are incremented using *IncMin*. Otherwise, the action terminates (**Skip**). If the event *time* occurs, then the values of min and sec are displayed (output), using the channel *out*, before termination.

*Circus* comes with a denotational and an operational semantics, based on Hoare and He's Unifying Theories of Programming (UTP) [19], and a notion of refinement. We can use *Circus* to write abstract as well as more concrete specifications, or even programs. A full account of *Circus* and its denotational semantics is given in [37].

The operational semantics [26] plays an essential role on the definition of testing strategies based on *Circus* specifications. It is briefly introduced below, and a significant part is reproduced in Appendix A. It is defined as a symbolic labelled transition system between configurations. These are triples $(c|s \models A)$, with a constraint $c$, a state $s$, and a continuation $A$, which is a *Circus* action. Transitions associate two configurations and a label. The

Please cite this article as: A. Alberto et al., Formal mutation testing for Circus, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.04.003