UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Formal mutation testing in *Circus* process algebra

Alex Donizeti Betez Alberto

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Alex Donizeti Betez Alberto

Formal mutation testing in *Circus* process algebra

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Adenilso da Silva Simão

Co-advisor: Prof. Dr. Marie-Claude Gaudel

USP – São Carlos
November 2018

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

A334f Alberto, Alex Donizeti Betez
 Formal mutation testing in Circus process
algebra / Alex Donizeti Betez Alberto; orientador
Adenilso Simão; coorientadora Marie-Claude Gaudel. -
- São Carlos, 2018.
 183 p.

 Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2018.

 1. TESTE E AVALIAÇÃO DE SOFTWARE . 2. ENGENHARIA
DE SOFTWARE . I. Simão, Adenilso, orient. II.
Gaudel, Marie-Claude, coorient. III. Título.

Alex Donizeti Betez Alberto

Teste de mutação formal aplicado na álgebra de processos
Circus

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Adenilso da Silva Simão

Coorientador: Prof. Dr. Marie-Claude Gaudel

USP – São Carlos
Novembro de 2018

*“Nobody exists on purpose, nobody belongs anywhere,
everybody’s gonna die. Come watch TV.”
(Morty, “Rixty Minutes”)*

RESUMO

ALBERTO, A. D. B. **Teste de mutação formal aplicado na álgebra de processos *Circus***. 2018. 183 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

ALGEBRAS de processos são uma família de técnicas de especificação e análise formal utilizadas em sistemas computacionais, especialmente em contextos de processos independentes, que atuam paralelamente e efetuam comunicação entre si. São os chamados sistemas concorrentes. *Circus* é uma álgebra de processos que agrega a capacidade de expressão de comportamentos concorrentes do CSP com a modelagem predicativa de dados da notação Z. Trabalhos recentes vêm estabelecer uma teoria para o teste simbólico exaustivo baseado em especificações modeladas em *Circus*. Com o objetivo de viabilizar a aplicação prática desses testes, é conveniente estudar critérios que reduzam o conjunto de casos de teste que, pela sua natureza exaustiva, torna-se frequentemente infinito. Neste sentido, o presente trabalho propõe a aplicação de técnicas de teste de mutação à partir de especificações *Circus*, visando a cobertura de falhas inseridas por meio de operadores de mutação já conhecidos, juntamente com operadores propostos especificamente para a linguagem. Algumas contribuições foram produzidas na busca destes objetivos, como o estabelecimento de uma teoria formal para a aplicação de teste de mutação em especificações *Circus* e a implementação de um gerador de rastros simbólicos para a mesma linguagem.

Palavras-chave: Teste de mutação, Especificação formal, Álgebra de processos, *Circus*.

ABSTRACT

ALBERTO, A. D. B. **Formal mutation testing in *Circus* process algebra**. 2018. 183 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

PROCESS algebras are a family of techniques used in formal specification and analysis of computer systems, specially when independent processes that perform and synchronize in parallel are concerned. The so-called concurrent systems. *Circus* is a process algebra that aggregates the expressiveness power for concurrent behaviors from CSP, along with the predicative data modelling aspects of Z. Recent publications have established a formal exhaustive symbolic testing theory for specifications modelled in *Circus*. Aiming to improve the feasibility of applying such tests in practical scenarios, it is convenient to look for criteria that reduces the number of test cases which, by the exhaustive nature of the approach, is often infinite. In the light of this, the work we present proposes the application of mutation testing techniques in *Circus* specifications, targeting the coverage of faults seeded by well-known mutation operators, along with operators designed with the particularities of the language in mind. Some contributions were produced in the pursuit of these goals, such as establishing a formal theory for mutation testing in *Circus* specifications and the implementation of a symbolic traces generator for the language.

Keywords: Mutation testing, Formal specification, Process algebra, *Circus*.

LIST OF FIGURES

Figure 1 – Sample CCS processes interacting	33
Figure 2 – Schema <i>MovingTrain</i> in <i>Z</i>	38
Figure 3 – <i>Circus</i> specification of a chronometer	39
Figure 4 – Examples of two transition rules: for inputs, and for guards	41
Figure 5 – Sample constrained symbolic traces for <i>Chrono</i>	42
Figure 6 – A mutant of the chronometer	63
Figure 7 – Syntax of specification labels.	66
Figure 8 – Transition rules that define specification traces	66
Figure 9 – Some specification traces for <i>Chrono</i>	67
Figure 10 – Some specification traces for <i>MutatedChrono</i>	67
Figure 11 – Operational semantics of labels	68
Figure 12 – Constrained symbolic traces for <i>MutatedChrono</i>	68
Figure 13 – Mutation of <i>Chrono</i> after dropping <i>time</i>	72
Figure 14 – Mutation of <i>Chrono</i> after dropping <i>tick</i>	72
Figure 15 – ERS Specification	79
Figure 16 – <i>ppoSeqInt</i> <i>ERSystem</i> mutants	80
Figure 17 – Mutations by <i>ppoParSeq</i>	81
Figure 18 – Overview of the tool chain implementation	85
Figure 19 – Part of an AST	86
Figure 20 – Class hierarchy for mutation operators	87
Figure 21 – AST changes by <i>OPRpcr</i>	88
Figure 22 – Interaction with the mutant generation utility	88
Figure 23 – Sample specification in <i>Circus</i> formal notation	89
Figure 24 – <i>Sample</i> specification in machine readable <i>Circus</i>	89
Figure 25 – Extract of <i>Sample eak</i> mutation	90
Figure 26 – CFG comparison for <i>Sample</i> and its <i>eak</i> mutant	91
Figure 27 – CFG for <i>Sample</i> example	92
Figure 28 – CFG sub-graph for sliced <i>Sample</i> example	93
Figure 29 – <i>Sample</i> sliced specification	95
Figure 30 – Solution output for mutated <i>Sample</i> sliced specification	95
Figure 31 – Overview of the configuration	103
Figure 32 – The process textual structures cell <i>procstrs</i>	104
Figure 33 – The process cell <i>procs</i>	105

Figure 34 – The process initial structures cell <i>inits</i>	106
Figure 35 – The process initial notification structures cell <i>oks</i>	106
Figure 36 – Sample graphical representation of a synchronization tree	108
Figure 37 – <i>Pump</i> specification in <i>Circus</i> formal notation	113
Figure 38 – <i>Pump</i> specification in machine readable <i>Circus</i>	114
Figure 39 – Extract from <i>Pump esk</i> mutant	115
Figure 40 – Extract of the CFG generated by the tool for <i>Pump esk</i> mutant	116
Figure 41 – <i>Pump</i> slice for <i>esk</i> mutant	116
Figure 42 – Symbolic execution comparison for <i>Pump esk</i> mutant	117
Figure 43 – Extract from <i>Pump pprSchema</i> mutant	117
Figure 44 – Fibonacci specification in <i>Circus</i> formal notation	118
Figure 45 – <i>Fibonacci</i> specification text	119
Figure 46 – <i>Fibonacci</i> eld mutant extract	120
Figure 47 – <i>Chrono</i> in CSP	125
Figure 48 – FDR output for $CHRONO \sqsubseteq_F MCHRONO$	126
Figure 49 – Two equivalent mutants of <i>Chrono</i> exposed by FDR	126
Figure 50 – Synchronization test specification	127
Figure 51 – Synchronization test output result	127
Figure 52 – Variable scope test specification	128
Figure 53 – Scope test output result	128
Figure 54 – Parallel state assignment specification text	129
Figure 55 – Parallel state assignment output result	129
Figure 56 – <i>Fibonacci</i> output result	130
Figure 57 – <i>Chrono</i> specification text	131
Figure 58 – <i>Chrono</i> output result	132
Figure 59 – Schema <i>BirthdayBook</i> in <i>Z</i>	147
Figure 60 – Operation schema for <i>AddBirthday</i>	149
Figure 61 – Schema for operation <i>RetrieveBirthday</i>	150
Figure 62 – Schema for the operation <i>Reminder</i>	151

LIST OF ALGORITHMS

Algorithm 1 – Finding data dependent nodes	92
Algorithm 2 – Finding control dependent nodes	94

LIST OF TABLES

Table 1 – Test case sets and their mutation score for Source code 1	58
Table 2 – Mutation operators for <i>Circus</i>	71
Table 3 – Analysis of mutants for <i>Pump</i>	115
Table 4 – Analysis of mutants for <i>Fibonacci</i>	119
Table 5 – Traces and states for <i>Fibonacci</i> mutants	120
Table 6 – Analysis of tool chain generated mutants for <i>Chrono</i>	122
Table 7 – Analysis of <i>Chrono</i> mutants in FDR	123

CONTENTS

1	INTRODUCTION	21
1.1	Contextualization	21
1.2	Motivation	23
1.3	Goals and contributions	23
1.4	Document organization	24
2	BACKGROUND	25
2.1	Considerations	25
2.2	Software development with formal methods	25
2.2.1	<i>Formalism in engineering</i>	25
2.2.2	<i>Formal specification</i>	28
2.2.2.1	<i>Process algebras</i>	29
2.2.2.2	<i>Communicating Sequential Processes (CSP)</i>	34
2.2.2.3	<i>Z notation</i>	37
2.3	<i>Circus language</i>	38
2.3.1	<i>Circus notation</i>	39
2.3.2	<i>Operational semantics</i>	40
2.4	Software testing	42
2.4.1	<i>Test and software quality assurance</i>	42
2.4.2	<i>Concepts and definitions</i>	43
2.4.3	<i>Testing criteria</i>	44
2.4.4	<i>Specification based test</i>	45
2.4.4.1	<i>Testing from process algebras</i>	46
2.4.4.2	<i>Testing in Circus</i>	51
2.4.5	<i>Mutation testing</i>	54
2.4.5.1	<i>Mutation operators</i>	55
2.4.5.2	<i>Equivalent mutants</i>	57
2.4.5.3	<i>Killing mutants</i>	58
2.5	Final considerations	60
3	FORMAL MUTATION TESTING IN <i>CIRCUS</i>	61
3.1	Considerations	61
3.2	Mutation testing in <i>Circus</i>	61

3.2.1	<i>Mutation against traces refinement</i>	62
3.2.2	<i>Circus mutation example</i>	62
3.2.3	<i>Mutation against conf conformance relation</i>	63
3.2.4	<i>Specification traces and mutation points</i>	65
3.3	Mutation operators	69
3.3.1	<i>Modification of behavioural operators</i>	70
3.3.1.1	<i>Mutations of events</i>	70
3.3.1.2	<i>Mutations of choice and concurrency operators</i>	73
3.3.1.3	<i>Mutations of communications</i>	75
3.3.1.4	<i>Mutations of name references and hiding</i>	76
3.3.2	<i>Expression modification operators</i>	76
3.4	Emergency Response System specification example	78
3.4.1	<i>Mutations by ppoParInt</i>	80
3.4.2	<i>Mutations by ppoParSeq</i>	80
3.5	Final considerations	82
4	TOOL DEVELOPMENT	83
4.1	Considerations	83
4.2	Mutation testing automation	83
4.2.1	<i>Circus mutant generation</i>	84
4.2.2	<i>Circus mutant analysis</i>	86
4.3	Symbolic evaluator for <i>Circus</i>	95
4.3.1	<i>The K-Framework</i>	96
4.3.2	<i>Syntax</i>	98
4.3.3	<i>Configuration</i>	102
4.3.3.1	<i>Structures of loaded processes</i>	103
4.3.3.2	<i>Local configurations for symbolic evaluation of processes</i>	104
4.3.3.3	<i>Handling of initials</i>	105
4.3.4	<i>Symbolic evaluation of process synchronizations</i>	107
4.3.5	<i>Conclusion and results</i>	110
4.4	Final considerations	110
5	EXPERIMENTS	111
5.1	Considerations	111
5.2	Tool chain validation experiments	111
5.2.1	<i>Methodology</i>	111
5.2.2	<i>Fuel Pump</i>	112
5.2.2.1	<i>Results analysis</i>	114
5.2.2.2	<i>Considerations</i>	117
5.2.3	<i>Fibonacci Generator</i>	118

5.2.3.1	<i>Results analysis</i>	118
5.2.3.2	<i>Considerations</i>	120
5.2.4	<i>Chronometer</i>	121
5.3	Theory validation by CSP refinement checking	121
5.3.1	<i>Translating Circus specifications into CSP</i>	123
5.3.2	<i>Using FDR for fault-based test-case generation</i>	124
5.4	Validating the symbolic execution tool	126
5.5	Final considerations	132
6	CONCLUSIONS AND FUTURE WORK	133
6.1	Contributions	133
6.2	Limitations	134
6.3	Future work	134
BIBLIOGRAPHY		137
APPENDIX A	Z SPECIFICATION EXAMPLE	147
APPENDIX B	IMPLEMENTATION OF K-FRAMEWORK REWRIT- ING RULES	153
B.1	Structural and embedding rules	153
B.1.1	<i>Specification processing and loading</i>	153
B.1.2	<i>Configuration maintenance</i>	157
B.2	Operational semantics rules	158
B.2.1	<i>Internal progress</i>	159
B.2.2	<i>State and Z schema</i>	162
B.2.3	<i>Observable progress</i>	164
APPENDIX C	CIRCUS OPERATIONAL SEMANTICS	175
C.1	Operational semantics rules	175
C.2	Specification traces operational semantics	180

INTRODUCTION

1.1 Contextualization

Research in recent decades has caused a conspicuous evolution of computational resources and, as a consequence, increased the potential for computers to be used as tools in a plethora of activities. Systems have become progressively complex, answering to the requirements of a growing set of responsibilities that has been assigned to them. At some point, computers were no more just tools that simply aid in increasing the efficiency of human tasks: nowadays they are trusted to perform activities that would be unfeasible without automation. It is fair to state that computer systems have become indispensable to the society.

As an effect of such process, computing is now applied in critical scenarios, where faults may lead to undesirable and even irreversible circumstances, such as risking human lives or causing severe material loss. Organizations like hospitals and stock markets support a substantial part of their operational structure over complex and high demanding computer systems. It is crucial for them to be trustworthy, motivating the search for manners of assuring that the behaviour of a system will be adequate under all designated situations.

To establish such trustworthiness, it is necessary to inquire about the quality of the components of a system. In that sense, *software engineering* provides methods and tools that aspire to make consistent the elaboration of such products. Among the many phases of an usual development process, one is of special interest when the correctness is at stake: *software testing*. Testing activities in a software project consumes significant part of the total development effort and, in the case of systems whose malfunctions may yield drastic consequences, the maximum attention for their verification is justifiable.([ADRION](#); [BRANSTAD](#); [CHERNIAVSKY](#), 1982)

Laid on the line, to test software consists in confronting the actual behaviour, exhibited by the final product, against the behaviour specified by design ([BOCHMANN](#); [PETRENKO](#), 1994). Beyond this simplistic description, the practical application of testing encloses several

caveats, resulting in an utterly complex task. In general, manual and interactive verification procedures are sufficient for small and simple modules. On the other hand, testing large and complex systems is unfeasible without automation. As a consequence, it is necessary to specify their behaviour in languages and models that may be subject to automated processing, without ambiguity. Thus, the usage of formal models to describe behaviours is specially adequate when dealing with a subgroup of systems that share specific characteristics, classified as reactive systems ([PRESSMAN, 2005](#)): systems that keep constant interaction with their environment by using atomic sequential input and output events. Such systems are applied for managing critical operations, e.g., traffic control, industrial plants, avionics, engines and general sensor monitoring and reacting activities ([HAREL; PNUELI, 1985](#)). Some examples of formal techniques used to specify reactive systems include: Finite State Machines (FSM), Statechart diagrams, Petri networks, Z language and process algebras, like CCS, CSP and π -Calculus.

Testing based on models, namely, model based test (MBT) ([HIERONS *et al.*, 2009](#)), is an approach that enables the automation of parts of the testing process, encouraging the design and application of efficient and reliable tests. MBT aims at checking whether an implementation under test is in compliance with its specification, benefiting from the precise and clear semantics of the formal models, as opposed to informal or semi-formal behaviour descriptions, subject to ambiguous interpretations.

Circus (Concurrent Integrated Refinement CalculUS) ([WOODCOCK; CAVALCANTI, 2001](#); [SAMPAIO; WOODCOCK; CAVALCANTI, 2002](#)) is a state-rich process algebra combining Z ([WOODCOCK; DAVIES, 1996](#)), CSP ([ROSCOE, 1998](#)), and a refinement calculus ([MORGAN, 1994](#)). Its denotational and operational semantics are based on the Unifying Theories of Programming (UTP) ([HOARE; JIFENG, 1998](#)), which uses relational calculus for defining denotational semantics to constructs of different programming paradigms. In the *Circus* theory, systems are characterized as processes, which group constructs that describe data and control behavior; the Z notation is used to define most of the data aspects, and CSP is used to define behavior. Moreover, the language provides support for the formal stepwise development of concurrent programs. The process algebra has already been used to verify, for example, software in aerospace applications ([CAVALCANTI; CLAYTON; O'HALLORAN, 2011](#)), and novel virtualization software by the US Naval Research Laboratory ([FREITAS; MCDERMOTT, 2011](#)).

Mutation testing is recognized as one of the most effective fault-detection techniques ([PAPADAKIS; MALEVRIS, 2010](#)). The systematic injection of feasible modeling faults into specifications allows the prediction of potential defective implementations. The faults are seeded by syntactic changes that may affect the observable specified behavior. Such faulty models are “mutants”. A mutant is “killed” by a test case able to expose its observable behavior difference. Testing can benefit from mutation in two ways ([JIA; HARMAN, 2011](#)): 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.

1.2 Motivation

A theory of testing for *Circus* (CAVALCANTI; GAUDEL, 2011), instantiating Gaudel’s long-standing theory of formal testing (GAUDEL, 1995; GAUDEL; JAMES, 1998; LESTIENNES; GAUDEL, 2002), is available. It is founded on the *Circus* operational semantics (WOODCOCK *et al.*, 2007), described and justified in the UTP (OLIVEIRA; CAVALCANTI; WOODCOCK, 2007). The yielded test sets are symbolic and composed of constrained symbolic traces, symbolic forbidden continuations, i.e., events that cannot happen after a particular trace of the specification, and symbolic acceptance sets. As expected, the exhaustive tests are infinite and, in practice, it is mandatory to rely on selection criteria both to generate and to select a feasible set of tests.

Model based test criteria is guided by testing requirements that should be met by a test suite. Usually, the requirements are either coverage criteria, stating 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. Testing is fault-based when its motivation is to demonstrate the absence of some given faults (MORELL, 1990). Mutation testing provides an objective model for exploring bounded fault-based testing sets.

Since we have coverage-based testing proposed for *Circus* in (CAVALCANTI; GAUDEL, 2014), it is relevant to investigate how fault-based techniques might be used to benefit practical testing for the process algebra. On these premises, to enhance the set of strategies for testing *Circus* by adding the possibilities of mutation testing is a step forward to advance the state of the art. Even though mutation testing has already been applied to languages and theories upon which *Circus* is based, e.g., CSP (SRIVATANAKUL *et al.*, 2003), the consideration of a state-rich process algebra for refinement with a UTP semantics is novel.

1.3 Goals and contributions

The main objective of this work is the formal definition for fault-based test sets, extending the *Circus* testing theory. We also propose an extensive study of mutation operators for the process algebra, considering fault-injection strategies based on previous works that have tackled similar challenges in related modeling languages (SRIVATANAKUL *et al.*, 2003; KUHN, 1999; BLACK; OKUN; YESHA, 2000).

The complementary goal is the construction of a chain of tools to handle *Circus* specifications, facilitating the automation of mutation testing procedures. In this regard, the present work comprehends the study of *Circus* operational semantics and the elaboration of heuristic algorithms for finding and exposing mutations by specification slicing.

The pursuit for such goals yielded the following contributions:

- Formal definitions for *Circus* mutation testing (ALBERTO *et al.*, 2016);
- Proposition and analysis of mutation operators, considering *Circus* particularities;
- Implementation of a symbolic evaluator based on *Circus* operational semantics. The evaluator is state aware and covers concurrency, allowing the inspection of internal symbolic execution states and producing constrained symbolic and specification traces (ALBERTO; GAUDEL, 2017);
- Design and implementation of tools to automate *Circus* mutation testing, exploring slicing techniques in mutant specifications to target the execution of seeded faults.

1.4 Document organization

In Chapter 2, a revision of theory and techniques over software engineering disciplines is presented. The selected topics are focused on formal specification and software testing, given that these subjects are the main concern of the present work. We also introduce the process algebra *Circus*, its underlying concepts and testing theory.

Chapter 3 contains the main contribution of this work. The formal mutation development for *Circus* is introduced, and we discuss how mutation testing relates to the already established testing theory for the process algebra. We present an extensive list of mutation operators, along with an analysis of their effects over the behaviour of specifications.

The development of tools is reported in Chapter 4, including considerations about the complexity involved in test automation. The links of the tool chain are described individually, as well as the heuristic algorithms designed to select traces from sliced specifications. Each module is presented with practical examples of its application. Considering the relevancy of the symbolic evaluator for *Circus*, its development is covered in details, providing descriptions of how the operational semantic rules relate to the implementation.

The Chapter 5 comprehends several experiments conducted to validate both the theory and the tool chain. It includes samples designed to stress the symbolic evaluator, experiments of mutation testing by refinement checking using CSP translations, and analysis of *Circus* specifications fed as input to the tool chain.

In Chapter 6, closure is provided to this document. We summarize the accomplished work, critically analyzing the outcome and recognizing eventual limitations. Improvements that should be targeted by future works are proposed as well.

BACKGROUND

2.1 Considerations

This chapter gathers information on topics that are paramount to the development of this work. A discussion about the use of formal methods in software engineering is presented in [section 2.2](#), introducing all the required background on formal software specification with process algebras and Z notation. Software testing is summarized in [section 2.4](#), presenting an overview of techniques and concepts such as specification based testing and mutation testing. In [section 2.3](#), we introduce the specification language and its testing theory, the targets of the present work.

2.2 Software development with formal methods

This section starts with a brief discussion, presented in [subsection 2.2.1](#), about the common software engineering practices, confronting the usual standing of software development methods in comparison with more traditional engineering realms. The application of formal methods in software development activities is also introduced in [subsection 2.2.1](#), followed by a presentation of concepts such as formal notations and process algebras in [subsection 2.2.2](#).

2.2.1 *Formalism in engineering*

The traditional engineering practices routinely deal with the complexity of mathematical models and, as a result, they benefit from the rigor of such models as foundations in project design. In an opposite sense, the disciplines of software engineering are inclined towards less precise approaches, shifting the focus to other fronts that prioritize the agility in prototyping to the detriment of precise and rigorous definitions.

In a typical software engineering project, natural language and informal diagrams are

largely adopted to define the specifications of systems (BOWEN; STAVRIDOU, 1993b). The use of formal methods is delayed until the implementation phase, when programming languages are finally applied. Overlooking such concerns is often justified by the complexity found in the mathematical modelling resources, with difficult notations, strict techniques that do not adapt well to the area needs or the lack of adequate tools to support and simplify its application (CLARKE; WING, 1996).

According to Bowen and Stavridou (1993a), software products are a reference in recurrence of errors, being the software industry famous for the imposition of use terms and responsibility agreements that would otherwise be unacceptable under the perspective of traditional engineering disciplines. One could attribute the lack of commitment to rigor to the flexibility that is inherent to the nature of software products, a realm where the corrections for design and implementation faults are, as a rule of thumb, distributed and applied with relative low cost and agility.

The discussion over the feasibility of using formal techniques in software development has been widely covered in literature (HALL, 1990; BOWEN; STAVRIDOU, 1993a; BOWEN; HINCHEY, 1995a; BOWEN; HINCHEY, 1995b; CRAIGEN; GERHART; RALSTON, 1995). Taking into consideration the application of the product, the reasons that support the use of such techniques may transcend the cost sheets: failures in critical systems might have ethical implications and the guarantees provided by formal methods shall be faced as mandatory (BOWEN, 2000).

Pressman (2005) classified software products in categories: basic software, information systems, scientific systems, embedded systems, personal systems, artificial intelligence systems and reactive systems. Concerning the use of formal methods, this last category calls for attention: a system is classified as reactive once it maintains continuous interaction with its external environment, be it an user, an input device or another system. Are examples of reactive systems: telephony, operating system modules, communication networks, automobile embedded technology, missiles and avionics (HAREL; PNUELI, 1985). As seem, the category of reactive systems covers applications that reside often in critical scenarios, where failures might cause irreversible critical situations, such as the loss of human lives or severe material damage. According to Harel (1987), the software engineering literature emphasizes the adversities in designing and specifying complex reactive systems. They arise from the difficulty of describing the reactive behavior in a clear and realistic way but, at the same time, formal and rigorous enough to allow such description to be evaluated under computer simulation.

According to Woodcock *et al.* (2009), formal methods are mathematical techniques, usually supported by tools, for the development of software and hardware systems. The use of such methods contributes to the quality of elaborated products, making the development processes less error prone (BARROCA; MCDERMID, 1992).

The mathematical rigor makes feasible the analysis and verification of models in any

phase of the life cycle in a development process: requirements engineering, specification, architecture, project design, implementation, testing, maintenance and enhancement. When adopted in premature phases, they aid in discovering project errors that would otherwise be hidden until the debugging activities. In later phases, formal methods help to check implementations equivalence (WING, 1990).

When designing specifications, formal methods allow the elaboration of precise statements to describe how a piece of software or hardware should perform. It is possible to write rigorous and, at the same time, abstract specifications, focusing in the desired behavior without paying attention to implementation minutiae. With the aid of formal methods, complex specifications might be decomposed in smaller segments that may be treated separately, without compromising the coherence. The sole fact of adopting a formal approach to describe a system is enough to rule out a substantial number of potential mistakes, as they are inherent to natural language (WING, 1990).

Large systems require a careful architectural organization of the components, emphasizing the convenience of using models that can suppress irrelevant details for a given development phase, allowing an architect to focus his efforts in satisfying requirements concerning the system overall structure (LAMSWEERDE, 2003). There are formal languages dedicated for such architectural descriptions, as in Allen (1997), based in formalizing the abstract behaviour of system components and their connecting links.

Formal techniques are also used to support project related activities, such as data refinement, by using finite state machines, abstraction functions and proof simulations (HOARE, 2002). Process algebras allows the precise definition of complex behaviour aspects, such as non-determinism and treating concurrency (HOARE, 1978).

On the implementation scope, formal methods might be applied to code verification. Each program-specification pair is implicitly related by a theorem that, once given conditions are satisfied, the specified and effective behaviors shall be identical (FLOYD, 1967). Code verification consists in the attempt of proving such theorem or, at least, determining the reasons for the impossibility of proving it. It is even feasible to automatically generate code from formal specification models, as shown by Abrial (1996), Freitas (2005) and Barrocas and Oliveira (2016).

Naturally, testing activities also takes benefits from the formal artifacts generated during the system development process. The automatic analysis of information represented in formal models allows the extraction and evaluation of test cases (CHOW, 1978; PENA; OLIVEIRA, 1998). The subject is covered in section 2.4.

2.2.2 Formal specification

Specification is the process of describing a system, stating its desired characteristics and properties (CLARKE; WING, 1996). According to Meyer (1985), specifications built over natural languages are prone to deficiencies, as classified below:

Noise: presence of textual noise that does not aggregate relevant information on the subject;

Omission: aspects not covered in the text;

Over specification: description of elements related to the solution, and not necessarily related to the problem;

Contradiction: conflicting definition of a same property or characteristic;

Ambiguity: descriptions that allows more than one interpretation;

Forward reference: reference to elements yet to be described later in the text;

Wishful thinking: descriptions of unrealistic elements, impossible to validate or even to implement in a real scenario.

To overcome the problems inherent to the use of natural language and incorporate the benefits cited in subsection 2.2.1, we can use formal methods to specify systems. A specification is said to be formal if it is written in a language that regards three sets of rules (LAMSWEEERDE, 2000):

Syntax: rules to grammatically check the well-formedness of the sentences;

Semantics: rules to precisely interpret the meaning of a sentence within a given domain;

Verification: rules to infer useful information about what is described.

The use of formal specifications has been considered by computer science disciplines since their origins: Turing observed that the analysis of sequential program was complemented by annotating properties in specific points of the execution flow and the set of values from such properties would compose a system internal state for the system (RANDELL; BABBAGE, 1973). By the end of the 60's, Naur and Randell (1969), Floyd (1967) and Hoare (1969) elaborated axiomatic techniques to prove the consistence between programs and properties.

Dijkstra (1975) shown how calculus could be used constructively to derive programs from abstract specifications, with mathematical guarantees for maintaining the described characteristics. Further techniques were also proposed to formally express specific needs, such as concurrent programs (PNUELI, 1977) and data structure based programs (PARNAS, 1983).

A plethora of formal models and languages were developed to aid in the task of specifying systems. The peculiarities of each technique make them more suitable to be used in certain domains. The fundamental differentiation between such formal techniques is based in its underlying paradigm (LAMSWEERDE, 2000):

History-based specification: considers the characterization of all the admissible behaviour sequences: the histories. Properties of interest are expressed in temporal logic assertions, relating past, present and future behaviours. These assertions are interpreted under a linear or concurrent time perspective, using discrete or continuous measurements. An history-based specification facilitates the imposition of conditions over temporal limits, what is convenient to state requirements of real time systems. The languages *ASTRAL* (GHEZZI; KEMMERER, 1991), along with others in the work of Katz, Richter *et al.* (1987), Moser *et al.* (1997), Jahanian and Mok (1986) are examples of formal techniques under this paradigm;

State-based specification: is based over the characterization of states which are desirable or admissible for the target system. Formal techniques founded in this paradigm usually explore the use of invariants and assertions. Invariants restrict the properties of the system, assuring that an observation taken at any arbitrary moment would reveal an admissible state. The assertions establish pre and post conditions to be taken into consideration when applying operations that modify the system state. Are examples of the state-based paradigm languages as Z (ABRIAL, 1980; SPIVEY, 1992; POTTER; TILL; SINCLAIR, 1996), VDM (JONES, 1990) and B (ABRIAL, 1996);

Transition-based specification: a specification based on transitions characterizes the changes between states of the system from input and output events. Properties are described by transition functions: given the current state and an input event, a new state and an output event are specified. The occurrence of an input event is considered sufficient condition for triggering a transition. Representations by Statecharts (HAREL, 1987), finite state machines (CHOW, 1978), Petri nets (PETERSON, 1977) and languages as PROMELA (HOLZMANN, 1991) are based on such paradigm.

The following subsections are dedicated to provide finer levels of details on some of the formal specification techniques that are foundations for a better understanding of the *Circus* language.

2.2.2.1 Process algebras

Process algebras are a family of techniques for formal description and analysis used in systems development. Techniques under such classification emphasize the specification

of independent components that work in parallel and interact with each other: the so-called concurrent systems.

It is convenient to describe the meaning of some usual terms within the context of process algebras, according to [Baeten \(2005\)](#):

System is a set of processes;

Process is any agent that produces observable behaviour;

Behaviour is all the events and actions that a system might perform, as well as the order in which they occur.

Examples of what can be modelled and analyzed under the perspective of a process algebra cover from the communication between software components to the behaviour of interacting people. The term *algebra* denotes the desired axiomatic approach: abstractions using symbols and the definition of formal rules for guiding their relationship.

Models such as automatas, that are adequate for describing and analyzing a wide spectre of behaviors, show deficiencies when concurrency is introduced. An important point is the lack of strategies to model interactions, i.e., communications between systems, processes, components or any other event generating agent. Such interactions are expected to affect the configuration in the final state. The notion of interaction is paramount for modelling distributed, parallel or concurrent systems, and process algebras are a theoretical approach designed for this end.

The foundations for elaborating the theory that underlies all process algebras are attributed to [Milner \(1980\)](#), who identified an algebraic structure in concurrent processes: starting from two processes P and Q , new processes might be built by combining the behaviors of P and Q in sequential or parallel forms. The resulting behavior of the combined process depends exclusively on the behaviors of both operand processes and the chosen combining operation. To build new elements from existing ones, according to a set of rules or axioms, is the notion behind the concept of algebra. As a conclusion, process algebras are algebraic perspectives for studying the behaviour of parallel systems, providing formal tools for their description and analysis ([BERGSTRÄ, 2001](#)).

Some mathematical definitions are common to any process algebra: three basic operators and seven fundamental axioms. They are presented next, although it is worth noticing that the notation will vary between different languages.

- “+”: alternative composition, i.e., $(x + y)$ behaves as x or y
- “.”: sequential composition, i.e., $x.y.z$ behaves as x , followed by y then z
- “|”: parallel composition, i.e., $(x | y)$ behaves as x and y simultaneously

The use of these operators allows the definition of complex compositions from a set of atomic actions. The axioms that rule how the operations relate to each other are:

- $x + y = y + x$ (alternative composition commutativity)
- $x + (y + z) = (x + y) + z$ (alternative composition associativity)
- $x + x = x$ (alternative composition idempotence)
- $(x + y).z = x.z + y.z$ (distributivity of sequential over alternative)
- $(x.y).z = x.(y.z)$ (sequential composition associativity)
- $x \mid y = y \mid x$ (parallel composition commutativity)
- $(x \mid y) \mid z = x \mid (y \mid z)$ (parallel composition associativity)

The precedence when combining alternative and sequential composition operators handles the priority to the operation ‘.’ over ‘+’. These axioms are considered static laws, since no execution aspects are made explicit. As a rule of thumb, any formal technique that considers these binary operators and the seven axioms to define the behavior of processes are said to be a process algebra. Next, two traditional process algebras are introduced along with more comprehensive examples of the basic operations. We begin with Calculus of Communicating Systems (CCS) (MILNER, 1980).

To handle a wide range of possible real world implementations for concurrent systems, the CCS process algebra deals with abstractions that may overlook a lot particularities but, on the other hand, provides a powerful and flexible way to describe interactions between parallel agents. There are no differentiation between active and passive agents, in a way that all elements of a CCS model are treated as processes. The interactions should be defined over synchronous message exchanges and more complex needs might be simulated.

In a CCS specification, a process is identified by a name, e.g., P , and it is considered a black box: all the interactions with a given process are defined through an *interface*, listing channel labels or ports. By convention, output labels are noted with a line over their names. As an example, a match stick is a *process* that waits for friction as input, then outputs heat. Next, a CCS model of such behaviour is shown:

$$PMatchStick \stackrel{def}{=} friction.\overline{heat}.0$$

Sequential Composition: given a label α , if P is a process, $\alpha.P$ is also a process that behaves as P after interacting in α .

The 0 at the end of the $PMatchStick$ process definition is a reference to the *null* process, i.e., the process that does nothing and is used to successfully end a process definition. The *null* behaviour is sequentially appended to the behaviour of the defined process using the sequential composition operator “.” (dot).

A match stick will not proceed until it gets an input in the *friction* port and, right after it, an output in the *heat* port is made available. Once consumed, the process stops by handling the control to the sequentially composed 0. Definition may also be made recursive, as in the *Clock* example:

$$Clock \stackrel{def}{=} \overline{tic}.\overline{tac}.Clock$$

After sequentially disposing the outputs in $\overline{tic}$ and $\overline{tac}$, the *Clock* process has a reference to itself, what causes it to restart and repeat the behaviour. As a result, the *Clock* process will continue in an infinite loop.

Alternative composition: if P and Q are processes, $P + Q$ is also a process that behaves either as P or as Q .

A process that contains an alternative composition has, initially, the potential to behave as any of the processes involved in the composition. What will actually define which behaviour will be followed is the sequence of presented inputs. When the sequence is not accepted by one or more of the composed processes, their behaviour is shutdown in a preemptive way.

Let *VendingMachine* be a process that behaves taking coins and dispensing coffee or tea. Coffee costs one coin and tea costs two coins. Next, a CCS model representing the specification of such a vending machine is shown:

$$VendingMachine \stackrel{def}{=} coin.(\overline{coffee}.VendingMachine + coin.\overline{tea}.VendingMachine)$$

After taking the first coin input, the process makes available a $\overline{coffee}$ output. If the output is consumed, the process is restarted and goes back to the original behaviour. Alternatively, if a second coin input is taken, the *VendingMachine* makes available a $\overline{tea}$ output before restarting.

Parallel composition: if P and Q are processes, $P \mid Q$ is also a process that behaves simultaneously as P and Q .

We now work on an oversimplification of the behaviour of a traditional software developer (*Programmer*): the production of lines of *code* conditioned by the consumption of *coffee*. The *Programmer* is allocated at an hypothetical *Company* that demands the production of lines of code. Given this scenario, we model in CCS the behaviour of the processes:

$$\begin{aligned} Programmer &\stackrel{def}{=} \overline{coin}.\overline{coffee}.code.Programmer \\ Company &\stackrel{def}{=} code.Company \end{aligned}$$

In order for these agents to collaborate, we need an external source that gives out *coffee*. So we model the following parallel composition using the previously defined *VendingMachine*:

$$Development \stackrel{def}{=} Programmer \mid VendingMachine \mid Company$$

A programmer needs *coffee* inputs to produce *code* outputs, *coffee* outputs are obtained from the *VendingMachine* agent, that consumes *coin* outputs from *Programmer*. As a result of the

coffee synchronization, i.e., matched input/output, the *Programmer* hands out the *code* outputs that are promptly consumed by the *Company*, at the other end of the composition. We used the tool PiVizTool (BOG; PUHLMANN; WESKE, 2007) to animate the sample CCS specification and the result is presented as an overview of the *Development* system in Figure 1.

The parallel composition defines concurrent execution of processes that might interact through known input and output labels. It is important to notice that these interactions are not mandatory: in the *Development* system example, the processes *Company* and *VendingMachine* never interact with each other, since they do not share any input/output label names.

Figure 1 – Sample CCS processes interacting

Source: Elaborated by the author.

Hiding operator: $P \setminus \{\alpha\}$ is a process that behaves as P , but all interactions through the channels listed in the set after the backslash are hidden, i.e., they cannot be observed by processes that are outside of the hiding operation.

The previous *Development* example might be rewritten exploring the behaviour of the hiding operator: the *Company* process interacts exclusively over *code* outputs, we may apply the channel hiding operator over the *Programmer* | *VendingMachine* composition and hide the interactions for *coin* and *coffee* labels.

$$Development \stackrel{def}{=} (Programmer \mid VendingMachine) \setminus \{coin, coffee\} \mid Company$$

Once the scope of those labels is restricted, the processes *Programmer* and *VendingMachine* behave like a single process with observable interactions only over *code* label. It is possible to model specifications based only in observable behaviors, abstracting the inner workings of parts of a system. We design a specification that would collaborate with *Company* as efficiently as the setup “ $(Programmer \mid VendingMachine) \setminus \{coin, coffee\}$ ”.

$$\begin{aligned}
\text{SpecDeveloper} &\stackrel{\text{def}}{=} \overline{\text{code}}.\text{SpecDeveloper} \\
\text{Development} &\stackrel{\text{def}}{=} \text{SpecDevelopment} \mid \text{Company}
\end{aligned}$$

Process equivalence: two processes *Spec* and *Imp* representing, respectively, a specification and a implementation, are equivalent in the sense that both perform the same external behaviour, despite the potential levels of abstraction or refinement between them. *Imp* correctly implements *Spec* if all behavioural aspects of *Spec* and no other unexpected behaviour are present in *Imp*.

The composition in *Development* process definition keeps its behaviour given that the process *Company* keeps getting *code* inputs. The source of such inputs does not matter: it might be the composition between a *Programmer* and his *VendingMachine* or any other process capable of generating $\overline{\text{code}}$ outputs. Let *Outsourcing* be a process capable of generating an infinite sequence of $\overline{\text{code}}$ outputs. Although we do not know exactly how *Outsourcing* works internally, its external behaviour matches the specification *SpecDeveloper* and might replace “ $(\text{Programmer} \mid \text{VendingMachine}) \setminus \{\text{coin}, \text{coffee}\}$ ” without any loss of functionality for the *Development* system.

The concept of process equivalence in CCS is unfolded into behavioural equivalence and observable equivalence. In a nutshell, the behavioural equivalence takes into consideration the number of internal transitions that a process performs between observable outputs. An internal transition is any interaction that is hidden, such as the *coin* and *coffee* exchange in the last example. We refrain from providing further details on CCS process equivalence as we move on to the next process algebra that has greater relevance for the goals of this work.

2.2.2.2 Communicating Sequential Processes (CSP)

When firstly introduced by [Hoare \(1978\)](#), the CSP stood as a programming language fragment, inspired by the guarded commands of [Dijkstra \(1975\)](#). Because of the minimalist and revolutionary approach for inter-process communication, the work gained space as a tool for the design and analysis of concurrent systems. In the upcoming years, CSP evolved with the refinement of concepts, becoming progressively suitable for building concurrent systems specifications. We present the language as it appears in [Hoare \(1985\)](#).

As in CCS, CSP processes are also identified by names. By convention, names are in upper case. The communication channels are generalized by the concept of *events*: atomic occurrences separated in time, identified by lower case names. For a process *P* that interacts through the events *a* and *b*, we say that the alphabet of *P* is $\alpha P = \{a, b\}$. Next, we show a CSP version of the match stick example that was used to introduce the CCS notation:

$$\text{MatchStick} = (\text{friction} \rightarrow \text{heat} \rightarrow \text{Skip})$$

The alphabet is $\alpha MatchStick = \{friction, heat\}$. There is no notion of complementary actions for events as they may occur in both directions: from the process to the environment and back. Two events synchronize when they are identified by the same label.

A *MatchStick* will hold until a *friction* event is made available by the environment. Immediately after its occurrence, a *heat* event is produced. The behaviour is concluded by invoking **Skip**: the basic process that only produces the $\checkmark$ event, representing the end of a successful execution. There is also the **Stop** basic process, used to suspend the execution or cause a *deadlock*: when one or more processes are attempting to synchronize but none of their input and output events correspond with each other (HOARE, 1978). The **Stop** process has an empty alphabet.

The CSP notation for recursive definition uses the syntax $\mu X : A \bullet P$, being A the event alphabet that causes the recursion. When $A = \alpha P$, it can be omitted from the expression. The CCS clock example is rewritten using such CSP notation for recursion :

$$\begin{aligned} Clock &= \mu X : \{tic, tac\} \bullet (tic \rightarrow tac \rightarrow X) \\ &= \mu X \bullet (tic \rightarrow tac \rightarrow X) \end{aligned}$$

Sequential composition: if P and Q are processes, $P; Q$ is also a process that behaves as P and, if P finishes successfully with a *Skip*, continues to behave as Q .

Guarded commands: if P is a process and g is a boolean expression, $g \ \& \ P$ is a process that behaves like P given that the boolean expression g is satisfied.

Prefixed communication: if e is an event and P is a process, $e \rightarrow P$ will proceed as P after a synchronization in the event e .

External choice: if P and Q are guarded or prefixed processes, $P \sqcap Q$ is also a process that behaves as P or Q . Once a guard or prefix is satisfied, the chosen process continues. If both are satisfied, the choice is non-deterministic.

Revisiting the vending machine CCS example that introduced the alternative composition, we present its CSP version using the external choice notation:

$$VendingMachine = \mu X \bullet coin \rightarrow (coffee \rightarrow X \sqcap coin \rightarrow tea \rightarrow X)$$

After synchronizing in the first *coin* event, two alternative paths are available: the *coffee* disposal or a new *coin* input. The next event will determine how the process continues, validating one of the prefixes. A more convenient way for enumerating alternative choice options is $(x : B \rightarrow P(x))$, given that B is a set of events and $P(x)$ is an expression that defines a process for each $x \in B$. We say that the set B lists the initial menu for the process, since it holds all the options that are suitable choices for selecting one of the branches.

Parallel composition: in CSP there are more complex parallel composition operators. We use the same processes used to exemplify the CCS parallel composition to present the notations:

$$\begin{aligned} \text{Programmer} &= \mu X \bullet (\text{coin} \rightarrow \text{coffee} \rightarrow \text{code} \rightarrow X) \\ \text{Company} &= \mu X \bullet (\text{code} \rightarrow X) \end{aligned}$$

The basic notation for parallel composition in CSP is similar to the CCS notation:

$$\text{Development} = \text{Programmer} \parallel \text{VendingMachine} \parallel \text{Company}$$

Since the notion of complementary events does not exist in CSP, the parallel processes interact when the names of events match in two or more processes. Given the composition $P \parallel Q$, synchronizations are expected for all event x that belongs to both alphabets: $\forall x \in \alpha P \cap \alpha Q$. We also have a modified versions of the parallel composition operator that allows the complete or partial isolation of the interacting alphabets. The notation $P \llbracket \{a, b\} \rrbracket Q$ restricts the synchronizations to the two events a e b , any other $x \in (\alpha P \cup \alpha Q) - \{a, b\}$ is isolated and do not synchronize within the composition. The *interleave* notation $P \parallel\!\!\parallel Q$ defines the complete isolation of both alphabets, not allowing any synchronization within the composition.

Channel hiding: As in CCS, there is a way to restrict the exposition of events to the outside environment. The syntax is also similar: $\alpha(P \setminus \{a, b\}) = \alpha(P - \{a, b\})$.

Communication channels: CSP provides a more elaborated way for inter-process communication via message passing. Given a process P , a channel c and a value x , the notation $(c.x \rightarrow P)$ defines a process that communicates x through the channel c and goes on behaving as P . Two parallel processes might exchange values using input/output channels identified with the same name.

By convention, a communication channel should be used for one way communication, i.e., exclusively for transmitting or receiving messages. With this in mind, a more specific notation is commonly used, depicting the input or output aspect of a channel: $(c?x \rightarrow P)$ defines the receiving of x in channel c and $(c!x \rightarrow P)$ denotes the transmission of x in the channel c .

Execution traces are presented in CSP with the notation $\langle a, b \rangle$, where a and b are two consecutive events. The set of traces of a process P , denoted as $\text{traces}(P)$, contains all the observable sequences of events that could be performed by P . The empty trace $\langle \rangle$ is part of any set of traces and is the only trace for the basic *Stop* process: $\text{traces}(\text{Stop}) = \{\langle \rangle\}$.

We present the set of traces for the *MatchStick* example process:

$$\begin{aligned} MatchStick &= (friction \rightarrow heat \rightarrow \mathbf{Skip}) \\ traces(MatchStick) &= \{\langle \rangle, \langle friction \rangle, \langle friction, heat \rangle, \langle friction, heat, \checkmark \rangle\} \end{aligned}$$

Next, we present the trace semantics for all basic processes and for the process compositions. These basic sets are the basis for analyzing semantics of more complex composition in an algebraic style (DAVIES, 2006).

$$\begin{aligned} traces(\mathbf{Stop}) &= \{\langle \rangle\} \\ traces(\mathbf{Skip}) &= \{\langle \rangle, \langle \checkmark \rangle\} \\ traces(a \rightarrow P) &= \{\langle \rangle\} \cup \{tr_P : traces(P) \bullet \langle a \rangle \frown tr_P\} \\ traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\ traces(P \sqcup Q) &= traces(P) \cup traces(Q) \\ traces(P; Q) &= \{tr_P : traces(P); tr_Q : traces(Q) \mid \\ &\quad (\checkmark \notin tr_P \wedge tr_Q = \langle \rangle) \vee tr_P \frown \langle \checkmark \rangle \in traces(P) \bullet tr_P \frown tr_Q\} \end{aligned}$$

The notation $\langle a \rangle \frown \langle b \rangle = \langle a, b \rangle$ indicates the concatenation of traces, unifying two traces in a single sequence of events.

Equivalence: the definition of equivalent processes in CSP is given by three distinct semantics: traces, failures and failures-divergence. The concept of refinement is used to compare processes under the perspective of one of the models. Due to the relevance of the topic for this work, the subject is treated with a more comprehensive approach in subsection 2.4.4.

2.2.2.3 Z notation

The Z specification language was introduced in University of Oxford (HAYES; FLINN, 1987; POTTER; TILL; SINCLAIR, 1996; SPIVEY, 1988; SPIVEY, 1992). The language is founded over the set theory and first order logic predicates. The mathematical basis in use is quite usual, except for some operators that were added aiming to yield simpler models.

A form of notation is employed to help structuring the specifications, since the extensive use of mathematical terms in large descriptions tends to complicate the organization, implicating in loss of intelligibility (BOWEN, 1996). The Z *schemas* are mathematical description sections that might be reused and combined using specific operators. Most of these *schema operators* is based over well known logic and math operations.

Besides formal descriptions, a Z specification may also include texts in natural language to aid in explaining the meaning of expressions. Ideally, the natural language sentences should be enough to describe the specification, even if the formal sections were missing. When there are

conflicts between what is specified in natural and formal forms, the formal description always takes precedence.

With some exaggeration, the Z notation may be considered as a functional programming language, describing the expected results instead of defining the means to achieve them. While a programming language is supposed to be exact and yield executable objects, a specification language is planned to maximize the power for expressing the requisites and stay intelligible to humans.

A Z specification is founded over three basic correlated constructs: **declarations** that introduce variables, **expressions** describing values that variables can assume and **predicates** that place constraints over such values. A Z schema combines these constructs for modeling, as shown in Figure 2, where the schema *MovingTrain* models simple aspects of a train moving in constant speed. There are three variables: *time?* for input, *distance!* for output and *speed* for internal calculation. The *speed* is set by the expression $speed = 60$ and the *distance!* output is calculated based on the input parameter *time?*.

Figure 2 – Schema *MovingTrain* in Z

<i>MovingTrain</i>
$distance!, speed, time? : \mathbb{N}$
$speed = 60$
$distance! = speed \times time?$

Source: Elaborated by the author.

Z schemas might be reused and combined to define complex data models, taking benefits from the set theory and function relations. In Appendix A, we added a full example that models a birthday book, including state and operation schemas. The example is presented in a tutorial approach and introduces further details that are fundamental to the data modeling aspects of the process algebra presented in the next subsection.

2.3 Circus language

Circus is a state-rich process algebra combining Z (WOODCOCK; DAVIES, 1996), CSP (ROSCOE, 1998), and a refinement calculus (MORGAN, 1994). Its denotational and operational semantics (WOODCOCK; CAVALCANTI, 2001; SAMPAIO; WOODCOCK; CAVALCANTI, 2002) are based on the Unifying Theories of Programming (UTP) (HOARE; JIFENG, 1998). *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 (CAVALCANTI; CLAYTON; O'HALLORAN, 2011) and novel vir-

tualization components by the US Naval Research Laboratory (FREITAS; MCDERMOTT, 2011).

We provide a comprehensive presentation of the *Circus* language in subsection 2.3.1 and its operational semantics in subsection 2.3.2. The *Circus* testing theory is introduced later, in section 2.4.

2.3.1 Circus notation

As exemplified in Figure 3, *Circus* allows us to model systems and their components via (a network of) interacting processes. In Figure 3, 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*.

Figure 3 – *Circus* specification of a chronometer

```

RANGE == 0 .. 59
channel tick, time
channel out : RANGE × RANGE

process Chrono ≡ begin
  state AState == [sec, min : RANGE]
  AInit == [AState' | sec' = min' ∧ min' = 0]
  IncSec == [ΔAState | sec' = (sec + 1) mod 60 ∧ min' = min]
  IncMin == [ΔAState | min' = (min + 1) mod 60 ∧ sec' = sec]
  Run ≡ tick → IncSec; ((sec = 0) & IncMin)
  □
  ((sec ≠ 0) & Skip)
  □
  time → out!(min, sec) → Skip
  • (AInit; (μ X • (Run; X)))
end

```

Source: Adapted from Oliveira (2006).

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 Figure 3, 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 Figure 3 defines *Chrono* itself.

Each process has:

a state and some operations for observing and changing it in a Z notation 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 $\bullet$. In our example, it is a sequential composition of the schema *AInit* followed by the repeated execution of the *Run* action. The *Circus* construct $\mu X \bullet A(X)$ 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'* for the state components after the operations. In each case, the seconds and minutes are incremented modulo 60.

Run starts with an external choice ($\square$) 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) (HOARE; JIFENG, 1998), 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 (OLIVEIRA; CAVALCANTI; WOODCOCK, 2009).

2.3.2 Operational semantics

The operational semantics (WOODCOCK *et al.*, 2007) 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 section C.1 of Appendix C. It is defined as a symbolic labelled transition system between configurations. These are triples $(c \mid s \models A)$, with a constraint *c*, a state *s*, and a continuation *A*, which is a *Circus* action. Transitions associate two configurations and a label. The 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.

The first component *c* of a configuration $(c \mid s \models A)$ is a constraint over symbolic variables that are used to define labels and the state. The constraints are texts that denote *Circus*

Figure 4 – Examples of two transition rules: for inputs, and for guards

$$\begin{array}{c}
\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xrightarrow{d?w_0} (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)} \\
\\
\frac{c \wedge (s; g)}{(c \mid s \models g \& A) \xrightarrow{\varepsilon} (c \wedge (s; g) \mid s \models A)}
\end{array}$$

Source: Elaborated by the author.

predicates over these symbolic variables. We use typewriter font for pieces of text. For example, $x := w_0$ is the text that describes an assignment of a value represented by a symbolic variable w_0 to the variable x . On the other hand, $x := w_0$ is the predicative relation that defines the meaning of $x := w_0$. The distinction is important when describing the operational semantics, where pieces of specification text are referenced along with the constraints.

The second component s is a UTP predicate, which defines a total assignment $x := w$ of symbolic variables w to all variables x in scope, including the state components. State assignments, however, can also include declarations and undeclarations of variables using the constructs $\text{var } x := e$ and $\text{end } x$. The state assignments define a value for all variables in scope. These values are represented by symbolic variables similarly to what is classically done in symbolic execution of programs (KING, 1976).

Two examples of rules are given in Figure 4. The first rule defines the transitions arising from an input prefixing $d?x : T \rightarrow A$; it is rule (C.1.5) of section C.1. The label of the transition is $d?w_0$, where w_0 is a symbolic variable. The constraint ruling that w_0 is of the right type ($w_0 \in T$) is added to the constraint of the new configuration. The state of the new configuration is enriched, via the UTP sequence operator “;”, by a new component x , which is assigned value w_0 . The continuation of the new configuration is the action A in an environment enriched by x as defined by $\text{let } x \bullet A$.

The second rule of Figure 4 defines the transitions arising from a guarded action $g \& A$. The label of such a transition is empty, since the evaluation of g is not an observable event; g is added to the constraint of the new configuration taking into account the assignments in the current state s . The continuation is A .

Traces of a process are defined in the usual way, that is, as sequences of observable events. Due to the symbolic nature of configurations and labels, however, we can obtain from the operational semantics *constrained symbolic traces*, or *cstraces*, for short. These are pairs formed by a sequence of labels, that is, a symbolic trace, and a constraint over the symbolic variables used in the labels. Roughly speaking, the constrained symbolic trace can be obtained

Figure 5 – Sample constrained symbolic traces for *Chrono*

$$\begin{aligned}
cst1 &: (\langle tick \rangle, true) \\
cst2 &: (\langle time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 0) \\
cst3 &: (\langle tick, time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 1) \\
cst4 &: (\langle tick \rangle^{60}, true) \\
cst5 &: (\langle tick \rangle^{60} \frown \langle time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 1 \wedge \alpha_1 = 0)
\end{aligned}$$

Source: Elaborated by the author.

by evaluating the operational semantics, collecting the labels together, and accumulating the constraints over the symbolic variables used in the labels. Figure 5 contains some examples of *cstraces* of *Chrono*. The notation $\langle e \rangle^n$ means the concatenation of an event e repeated by n times.

A trace is an instantiation of a *cstrace*, where the symbolic variables used in the labels are replaced by values satisfying the constraint. For instance, the two traces $\langle time, out!0!0 \rangle$ and $\langle tick, time, out!0!1 \rangle$ are instances of *cst2* and *cst3*.

2.4 Software testing

This section contains general information and concepts on the software testing theory and practice. Initial considerations about the role of testing activities in a development process are presented in subsection 2.4.1. In subsection 2.4.2 we introduce basic terms and the mandatory concepts for the discussions on the subject. Types of testing criteria and test techniques are presented in subsection 2.4.3. Specification based test is introduced in subsection 2.4.4. We present the foundations and some further discussions on mutation testing in subsection 2.4.5.

2.4.1 Test and software quality assurance

Dijkstra (1972) stated that testing software can only point out the presence of faults, however it can never assure their nonexistence. The observation is incontestable, considering the several limitations that affect software testing activities and more generally software verification. Some fundamental questions that are paramount to such activities are considered undecidable from the theory of computation perspective, including: to determine the equivalence between programs; to determine if the fragment of a program is executable; to determine if two sequence of commands compute the same function (MALDONADO *et al.*, 2004).

A way to assure the correctness of a program is, instead of submitting it to tests, to present a formal proof of its behaviour (NAUR; RANDELL, 1969). A number of practical reasons renders this approach unfeasible in real world situations, e.g., the requirement of having

the whole environment where the software is inserted to be available in an axiomatic form (GOODENOUGH; GERHART, 1975).

Considering the impossibility of guaranteeing the total absence of faults in a piece of software, we look for ways to reduce their incidence to a minimum. Techniques for software quality assurance (SQA) lead the development processes to more consistent and systematic practices. For the scope of this work, we have special interest in verification, validation and testing (VV & T) activities.

Verification aims to assure that all software features are correctly implemented, while validation checks that such features perform accordingly to their requisites (PRESSMAN, 2005). It is the testing activity that searches for previously undetected faults, what is usually accomplished by executing the software or parts of it (MYERS, 2004). Testing implies that some kind of executable implementation is available.

2.4.2 Concepts and definitions

To avoid interpretation mistakes and improve the exchange of information, allowing the knowledge sharing between those interested in software testing, the IEEE established the standard 610.12-1990 (IEEE, 1999). It is a glossary for the testing terminology in software engineering that makes the following differentiation:

fault: an incorrect element in the software, such as wrong steps, processes or data definition;

mistake: is the action taken by someone involved in the development process that led to spawning a fault;

error: the difference between a value produced by a faulty implementation and the expected value, according to the specification;

failure: is the exhibition of an error, usually due to the manifestations of a fault.

Assembling these concepts in a linear way: tests are applied to search for failures that are caused by errors, which are risen by faults, potentially inserted by mistakes. In the same line of thought, we might separate causes and effects: faults and failures are caused by mistakes and errors, respectively. In the scope of this document, we use the terms *error* and *failure* to allude reasons and consequences of software issues.

It is desirable to apply testing activities in all levels of a development process. Once product fragments are available in executable form, techniques may be used to search for errors. According to Pressman (2005), testing in different levels of a development process can be classified as follows:

Unity test: by unity, we refer to the smallest executable fragment of a software. The definition of unity depends on the context. For instance, in object oriented programming, a unit might be a method, a class, a component or any other kind of module. The individual test of units helps to detect faults in the implemented logic, allowing the developers to take the correcting measures in early phases.

Integration test: unity interface checking. Assuming that the correct behaviour of all units is already checked by the previous testing activities, integration test can be used to evaluate how they behave when interacting with each other. System features that involve the collaboration of two or more units are verified in this level.

System test: the system test is applied in the late phases of development and it aims to evaluate the product in a broader approach. All elements that compose the environment where the system will be inserted should be taken into consideration. It might even involve the execution platform and external systems as they interact with the product being tested.

A specification S is a description, formal or not, of how a given program P should produce outputs when elements of the domain D are presented as inputs. The P program works correctly if the produced output for all elements of D is in harmony with what has been described in S , e.g., $\forall x \in D, S(x) = P(x)$. The manifestation of a failure occurs by the production of an output that has not been specified in S , indicating that P is in an error state. A *test case* is an ordered pair $(x, S(x))$, where $x \in D$, that consists in an element x belonging to the domain D , along with an output that, according to S , should be produced by P when x is presented as input.

To expose all the failures that a software might manifest, it is natural to consider testing it using as inputs the whole domain D , checking each produced output against what is defined by S . In other words, it is the same as assembling and executing test cases for all elements in D . However, real world domains are often infinite or hold a very large element count (FRANKL; WEYUKER, 2000). Facing the impossibility of testing with the whole domain, it is necessary to select a subset $T \subseteq D$ with a size small enough to make testing feasible.

The subset T should be as representative as possible in relation to D , culminating in the condition $\forall t \in T, P(t) = S(t) \Rightarrow \forall x \in D, P(x) = S(x)$, e.g., it would be enough to test P with all elements in T to assure its correctness. Although a T subset with such a property always exists, there is no systematic way to build it (BUDD; ANGLUIN, 1982). It is a task similar to deciding the equivalence between programs, that we already mentioned in subsection 2.4.1 as one of the undecidable problems that support the discouraging Dijkstra statement on testing.

2.4.3 Testing criteria

Since it is generally impossible to systematically select a feasible number of elements for $T \subseteq D$ that would make the test cases as effective as testing from the whole domain D ,

strategies can be applied to select a relatively small number of elements from D in order to maximize the exposition of failures under some given aspect. These strategies are known as *testing criteria*. A criterion should aid in testing activities by providing answers for two questions: which elements should be selected from D and how effective is the resulting T subset in testing P (MALDONADO *et al.*, 2004). Although related, the questions should be treated separately using two distinct procedures (BUDD; ANGLUIN, 1982):

Generator: procedure $G_C(P, S)$ that, given a P program and its specification S , generates a set T adequate to test P according to the criterion C .

Acceptor: given P , S and a the subset T , the procedure $A_C(P, S, T)$ returns a true value when T is adequate to test P according to the criterion C .

The criteria is subject to certain classifications, according to the source under analysis: *specification based test* and *program based test* (HOWDEN, 1986). It is also usual to find names as *structural test* and *functional test* that refers to a similar classification. Specification based test, or structural test, are more relevant in the scope of this work, and is presented in finer details in the next subsection.

2.4.4 *Specification based test*

Testing criteria based on specifications take into consideration the description of the software functions, as they are presented in the specification models and documents, and use such information to build the set of test cases. Implementation details are ignored to the point of considering the program under test as a black box of unknown content: only the inputs and outputs can be asserted. It is rather common to see the term *black box testing* being applied to describe specification based testing practices (MYERS, 2004).

By not considering the implementation details, functional testing criteria might be applied without further concerns with the context of development of the product being tested. With this in mind, a set of test cases selected by a specification based criterion is, in theory, adequate to test either an implementation built under the procedural paradigm, either an objected-oriented version of it.

The main limitation of specification based approaches is the wide adoption of informal specifications in software development processes. The lack of formalism adds a substantial level of complexity for the automation of test case generating procedures and the fault coverage achieved by the testing activity is imprecise as well (PERRY; KAISER, 1990). The selection of test cases from natural language specifications is accomplished manually (with very few and specific exceptions). On the other hand, the use of mathematical models for describing systems allows the automation of some part of the test activities. Since the scope of this work is related to test based in process algebra specifications, the next subsection is dedicated to cover the subject.

2.4.4.1 Testing from process algebras

As programs, the behaviour of processes can also be investigated by tests (NICOLA; HENNESSY, 1984). Given a process specification SP and a system under test SUT , a testing activity may be presented as a conformity relation in the form $SUT \text{ sat } SP$ (CAVALCANTI; GAUDEL, 2007). The target of the tests in this context is an executable implementation that behaves as a dynamic entity. The interaction with such entity is subject to the limitations of exposed interfaces, often restricting observation and control.

Test cases for concurrent processes are *testing processes*: processes built to interact with the SUT using a shared alphabet. A parallel composition involving the SUT and the testing process is used to explore the specified behaviour and expose potential implementation faults. The final verdict is based on the observation of the generated output, taking as perspective a satisfaction relation (CAVALCANTI; GAUDEL, 2007).

Most of the methods for process algebra based testing take into consideration two kinds of observations: external events and deadlock. Deadlock is observed via time-out event triggers: if an implementation stops reacting after a given time limit it is said to be in a deadlock. The testing processes are built based on the specification considering the chosen satisfaction relation. Additional information about the SUT inner workings, the environment or even general assumptions are said to be the *testability hypothesis*.

The use of formal techniques results in unbiased tests, with objective methods to assert their efficacy. These aspects culminate in the definition of *complete* or *exhaustive* tests (GAUDEL, 1995): such tests are assured to approve all the implementations that satisfy the specification and reprove any other. An implementation that succeeds in an exhaustive test session is a reference of correctness (BERNOT; GAUDEL; MARRE, 1991).

An exhaustive test approach takes as basis the semantics of the formal specification. In most cases, it is unfeasible for practical use due to infinite input domains and non-determinism in the models. For practical use, a formal test criterion can be applied to select a subset of the exhaustive tests while holding certain properties. The formalism allows an objective evaluation for the loss of covering due to the reduction in the number of test cases (PELESKA; SIEGEL, 1996). Next, we introduce an exhaustive testing technique for the CSP process algebra.

CSP exhaustive testing

The work from Cavalcanti and Gaudel (2007) established solid foundations for model based testing using CSP specification. The testability hypothesis along with its consequences have been characterized. The exhaustive test set was formalized and validated by algebraic proofs from the language semantic definitions.

Refinement and semantics: To formally represent the behaviour of CSP processes, three denotational semantics models were defined and prioritized according to their level of expressiveness. The basic *traces model* (T) considers as behaviours only the execution traces. Analyzing a process behaviour from this model perspective will not allow the distinction from a deterministic to a non-deterministic behaviour, or even identify a deadlock situation.

In the next level of expressiveness we have the *failures model* (F), that adds rejection sets to the traces models. For each valid execution trace, there is a set of events that are not expected as valid continuations for the process under analysis. The failures model is enough to deal with the two aspects ignored by the traces model. The third and most expressive semantic model is the *failures-divergences* (FD), that captures the possibility of internal loops that might lead the process to a *divergent* state, i.e., an internal unrestricted sequence of non-observable actions (ROSCOE, 2008). The set $divergences(P)$ is the set of traces from P that lead to divergence, i.e., an infinite sequence of internal events and their extensions.

The notion of *refinement* is related with the compatibility between processes. We say that $P \sqsubseteq Q$, i.e., the process Q refines the process P , given that a refinement relation can be observed. Considering the three aforementioned semantic models for CSP processes, we show the consequent three refinement relations:

$$P \sqsubseteq_T Q \text{ - traces refinement, } traces(Q) \subseteq traces(P)$$

$$P \sqsubseteq_F Q \text{ - failures refinement, } failures(Q) \subseteq failures(P)$$

$$P \sqsubseteq_{FD} Q \text{ - failures-divergences refinement, } \\ failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P)$$

The first refinement relation implies the nonexistence of observable behaviour in the refined process that is not also observable in the original process. As a consequence, the basic process **Stop** is a refinement of every possible process, since the empty trace is, by definition, part of the alphabet of all CSP processes.

The failures refinement relation keeps the behaviour restriction from the first relation and aggregates the observation of rejection aspects: the refining process must only reject events that are also rejected by the original process. The mandatory and forbidden behaviours are listed using the set *failures*, where $failures(P)$ is a set of ordered pairs (t, A) with the subset of rejections A , for the process P after following the trace $t \in \Sigma^*$, given that Σ contains all events with the exception of $\checkmark$.

$$traces(P) = \{t : \Sigma^{\checkmark*} \mid (t, \emptyset) \in failures(P)\}$$

As the expressiveness of the semantic model used for behaviour comparison increases, more compatibility restrictions weight over the involved processes. The traces refinement is the

weakest of the relations. On the other side of the scale, the failures-divergences is the strongest. The notion of process equivalence is defined using refinement relations: we say that $P = Q$, i.e., a process P is equivalent to a process Q , when both refinement relations $P \sqsubseteq_{FD} Q$ e $Q \sqsubseteq_{FD} P$ hold.

Refinement and testing: The presented refinement relations are a starting point for CSP based test: assuming as part of the testability hypothesis that the *SUT* behaves as a CSP process, it is a testing activity to check if the hypothetical *SUT* process is a refinement of the known CSP specification.

Traditionally, the research for formal testing techniques are based in labelled transition systems and finite state machines (CAVALCANTI; GAUDEL, 2007). In a process algebra context, it is necessary to establish a formal link between the denotational and operational semantics in order to create a feasible connection between testing and refinement. The refinement relations based in denotational semantics can be trivially defined over subset inclusion statements. As for testing matters, it is mandatory to reference operational semantics. For example, a transition cover testing criterion requires the explicit notion of states and transitions in order to be formulated.

We say that a system P behaves as a CSP process Q when the execution of P or Q yields a similar set of observations, in any environment or context. The equivalence is given by checking the refinement in both directions, abstracting the fact that P is not an actual CSP process. In a more formal notation, if SUT_{CSP} represents a hypothetical CSP process that behaves like *SUT* and SP is a CSP modelled specification, we must check for $SP \sqsubseteq_F SUT_{CSP}$. The choice for failures refinement as the satisfaction relation is pointed in the work of Peleska and Siegel (1996). The failures-divergences model is not considered in testing under the assumption that specifications and implementations are divergence free (CAVALCANTI; GAUDEL, 2007) and the testability hypothesis also requires events to be observable as atomic, i.e., of irrelevant time duration.

Testing for traces refinement Since this satisfaction relation demands $traces(SUT) \subseteq traces(SP)$, but not the contrary, testing for traces refinement does not require executing traces of SP in *SUT*. The testing strategy must use the traces $\forall t \in \alpha SP^{\checkmark*}$ and $t \notin traces(SP)$, checking that they are rejected.

Actually, in a more concise strategy, it is enough to check only the minimal prefixes of forbidden traces, i.e., if after the trace $\langle a, b \rangle$ there is a forbidden event c , every trace prefixed by $\langle a, b, c \rangle$ is also forbidden and shall not be checked. Next, the formal definition for the traces testing set:

$$\{s \frown \langle a \rangle \mid s \in traces(SP) \wedge a \notin initials(SP/s)\} \quad (2.1)$$

Where SP/s denotes the behaviour of process SP after executing the trace s . The set $initials(P/s)$ holds all events specified as valid continuations for P immediately after executing s .

The set can be defined for any P process as $initials(P) = \{a : \Sigma^\vee \mid \langle a \rangle \in traces(P)\}$. For a test to be succeed, the following aspects must be observed:

- If a trace s followed by a deadlock is observed, the test is successful;
- If a trace $s \frown \langle a \rangle$ is observed, the test fails;
- If a prefix of s followed by a deadlock is observed, the test is said to be *inconclusive*: the trace s cannot be verified in the *SUT*, what is acceptable considering the traces refinement definition.

Assembling the test set using ordered pairs (s, a) , where $s \in traces(SP)$ and $a \in initials(SP)$, takes as reference the work of [Peleska and Siegel \(1996\)](#). The approach is enough to detect all trace refinement violations and also results in the minimal set that holds such property.

As described for in [subsubsection 2.4.4.1](#), testing in the realm of processes algebras consists in a parallel execution involving the *SUT* and a tester process T , resulting in $(SUT \parallel [\alpha SP]T) \setminus \alpha SP$ for CSP. The synchronization set αSP , which is also the interface between the processes, is hidden from the outside context. Such isolation is necessary to avoid external interferences from the environment that might affect the synchronizations inside the testing parallel composition. On the other hand, there are no observable outputs to check the test result. With this in mind, the tester process is instrumented with three special evaluation events: *pass*, *fail* and *inc*. They represent, respectively, the verdicts for a successful, unsuccessful and inconclusive test.

Given a finite trace $s = \langle s_1, s_2, \dots, s_n \rangle$ and an event a , the definition of a tester process is:

$$\begin{aligned}
 T_t(s, a) &= inc \rightarrow s_1 \rightarrow inc \rightarrow s_2 \rightarrow inc \dots s_n \rightarrow pass \rightarrow a \rightarrow fail \rightarrow \mathbf{Stop} \\
 T_t(\langle \rangle, a) &= pass \rightarrow a \rightarrow fail \rightarrow \mathbf{Stop} \\
 T_t(\langle \checkmark \rangle, a) &= pass \rightarrow a \rightarrow fail \rightarrow \mathbf{Stop} \\
 T_t(\langle b \rangle \frown s, a) &= inc \rightarrow b \rightarrow T_t(s, a)
 \end{aligned}$$

The definition of the exhaustive test set for a CSP specification SP is built over such concepts:

$$Exhaust_T(SP) = \{T_t(s, a) \mid s \in traces(SP) \wedge a \notin initials(SP/s)\}$$

The exhaustiveness of the defined set is formally demonstrated by [Cavalcanti and Gaudel \(2007\)](#), providing guarantees that a *SUT* approved by the tests from $Exhaust_T$ is, indeed, a traces refinement of its specification SP .

Testing for failures refinement: Checking the failures refinement via tests involves to assure that, after the execution of a common trace between SP and SUT , the observable failures of the SUT are also failures of SP . In that sense, it should be checked that, after all trace $\forall s \in \text{traces}(SP)$, the SUT shall refuse any events from the set $A \subseteq \alpha SP$, as they are not defined in SP after t .

The test run uses a tester process for the trace s , followed by an external choice offering all events in A , according to the definition $s; (\Box a \in A \bullet a \rightarrow \mathbf{Stop})$, given that $s \in \text{traces}(SP)$ e $(s, A) \in \text{failures}(SP)$. The tests may yield the following verdicts:

- If the trace s followed by a deadlock is observed, the test failed;
- If the trace $s \hat{\ } \langle a \rangle$, with $a \in A$, is observed, the test succeeded;
- If a prefix of s followed by a deadlock is observed, the test is inconclusive.

With the aid of the events *inc*, *pass* and *fail*, as they were used for instrumenting the verdicts in traces refinement testing, we can define the failures refinement tester process: given a finite trace $s = \langle s_1, s_2, \dots, s_n \rangle$ and a set A such that $(s, A) \in \text{failures}(SP)$:

$$T_f(s, A) = \text{inc} \rightarrow s_1 \rightarrow \text{inc} \rightarrow s_2 \rightarrow \text{inc} \dots s_n \rightarrow \text{fail} \rightarrow (\Box a \in A \bullet a \rightarrow \text{pass} \rightarrow \mathbf{Stop})$$

$$T_f(\langle \rangle, A) = \text{fail} \rightarrow (\Box a \in A \bullet a \rightarrow \text{pass} \rightarrow \mathbf{Stop})$$

$$T_f(\langle a \rangle \hat{\ } s, A) = \text{inc} \rightarrow a \rightarrow T_f(s, A)$$

It is possible to optimize the test set considering that, if $(s, A) \in \text{failures}(SP)$, then $\forall A' \supset A \bullet (s, A') \in \text{failures}(SP)$. Once a SUT is successful in a test for A , it should also succeed for A' . On the other hand, if a test fails for A , the SUT could yet succeed when testing with A' , yet a fault was already revealed. For each trace s , it is sufficient to consider a subset of the sets A' where $(s, A') \notin \text{failures}(SP)$. Given a set A_s of sets $A_1, \dots, A_m$, $\forall A_i, (s, A_i) \notin \text{failures}(SP)$ e $\forall (s, A) \in \text{failures}(SP), \exists A_i \in A_s \mid A \supseteq A_i$.

$$\text{Exhaust}_{\text{conf}}(SP) = \{T_f(s, A) \mid s \in \text{traces}(SP) \wedge A \in A_s\}$$

$$A_s = \min_{\subseteq} (\{A \mid (s, A) \notin \text{failures}(SP)\})$$

$$\min_{\subseteq}(S) = \{S \mid S \in S \wedge \nexists S' \bullet S' \in S \wedge S' \subset S\}$$

The exhaustivity of the test set $\text{Exhaust}_{\text{conf}}$ was proven by [Cavalcanti and Gaudel \(2007\)](#). The authors provided formal assurances that a SUT , with no traces resulting in deadlock after a *fail* verdict, is in conformity with the specification SP .

Test case selection criteria: As discussed in [subsection 2.4.3](#), it is necessary to select finite subsets from an exhaustive test set. Strengthening the testability hypothesis is one strategy used for such end, introducing the so-called selection hypothesis. A weak selection hypothesis yields extensive test sets, while stronger hypothesis yields convenient small sets, but with limited applicability.

Several selection hypothesis might be formulated and combined based on the knowledge about the *SUT* internal structure, criteria for specification coverage and considerations on overall test application cost. For instance, a regularity hypothesis may establish that a *SUT* that succeeds on all tests of a given maximum size, will also succeed on the remaining elements of the exhaustive test set. In the context of traces, the number of events is the measure for test case length.

An uniformity hypothesis establish that a *SUT* will behave uniformly within given subsets of the exhaustive test set and, for that reason, it would be enough to test using an unique element from each of these subsets. An example of application for this hypothesis is the transition cover criteria on finite state machines ([FUJIWARA et al., 1991](#)). This sort of hypothesis is suitable for creating test case generators for finite models.

In the context of process algebras, it is possible to reference the operational semantics in order to derive labelled transition models. For instance, CSP models can be translated as state machines and binary decision diagrams ([PARASHKEVOV; YANTCHEV, 1996](#)). Although often infinite, criteria may be adapted from the finite state models testing theory.

The testing theory for CSP from [Cavalcanti and Gaudel \(2007\)](#) does not benefit from finite models selection strategies such as transition cover. With that in mind, the authors suggest an investigation for more adequate uniformity hypotheses.

2.4.4.2 Testing in Circus

A theory of testing for *Circus* ([CAVALCANTI; GAUDEL, 2011](#)), instantiating Gaudel's long-standing theory of formal testing ([GAUDEL, 1995](#); [GAUDEL; JAMES, 1998](#); [LESTIENNES; GAUDEL, 2002](#)), is available. It is founded on the *Circus* operational semantics ([WOODCOCK et al., 2007](#)), described and justified in the UTP ([OLIVEIRA; CAVALCANTI; WOODCOCK, 2007](#)). 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.

For divergence-free models, the *Circus* refinement relation can be characterized by the conjunction of traces-refinement and the well known *conf* relation, as defined in ([BRINKSMA, 1988](#)), that requires reduction of deadlocks. This is proved in ([CAVALCANTI; GAUDEL, 2010a](#)). Accordingly, ([CAVALCANTI; GAUDEL, 2011](#)) defines separate exhaustive test sets for traces

refinement and *conf*, namely $Exhaust_T(SP)$ and $Exhaust_{conf}(SP)$, which are presented next.

A test for traces refinement is constructed by considering a trace of the *Circus* specification and one of the events that cannot be used to extend that trace to obtain a new trace of the *Circus* specification (CAVALCANTI; GAUDEL, 2007). Such events are called the forbidden continuations of the trace. For a specification SP , the exhaustive test set $Exhaust_T(SP)$ includes all the tests formed by considering all the traces and all their forbidden continuations and inserting the special verdict events, as explained below.

For a finite trace $s = \langle a_1, a_2, \dots, a_n \rangle$ and an event (forbidden continuation) a , we define the test process $T_T(s, a)$ as follows:

$$T_T(s, a) = inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow inc \dots a_n \rightarrow pass \rightarrow a \rightarrow fail \rightarrow STOP$$

As introduced for CSP testing, extra special events *inc*, *pass* and *fail* are used to indicate a verdict. While executing a testing experiment, the test case is run in parallel with the SUT and the last special event observed in the trace provides the verdict. Due the possibility of nondeterminism, the submitted trace of the *Circus* specification is not necessarily performed by the SUT. The *inc* event indicates an inconclusive verdict: the SUT has not performed the proposed trace. If it does perform the trace, a *pass* event is observed, but if the SUT proceeds to engage in the forbidden continuation a , then there will be a *fail* event.

For instance, a possible test of *Chrono*, based on the trace $\langle tick, time \rangle$ and the forbidden continuation *out.0.0* is:

$$inc \rightarrow tick \rightarrow inc \rightarrow time \rightarrow pass \rightarrow out.0.0 \rightarrow fail \rightarrow Stop$$

This leads to the following definition of the exhaustive test set for traces refinement:

$$Exhaust_T(SP) = \{ T_T(s, a) \mid s \in traces(SP) \wedge s \frown \langle a \rangle \notin traces(SP) \}$$

The exhaustivity, that is, the equivalence of traces refinement to the absence of *fail* verdict when running all the tests of $Exhaust_T(SP)$, is proved in (CAVALCANTI; GAUDEL, 2007) and (CAVALCANTI; GAUDEL, 2011) under the *complete testing assumption* (FUJIWARA; BOCHMANN, 1991): when a test experiment is performed a sufficient number of times all possible (nondeterministic) behaviours of the SUT are observed.

Traces and forbidden continuations are characterized symbolically leading to the definition of $SExhaust_T(SP)$, an exhaustive set of symbolic tests based on cstraces and constrained symbolic forbidden continuations. An example of such a symbolic test for *Chrono*, based on the cstrace $(\langle tick, time \rangle, true)$ and the forbidden constrained symbolic continuation defined as $out.\alpha_0.\alpha_1 : (\neg(\alpha_0 = 0 \wedge \alpha_1 = 1))$ is composed as follows:

$$\begin{aligned} & inc \rightarrow tick : true \rightarrow inc \rightarrow time : true \rightarrow pass \\ & \rightarrow out.\alpha_0.\alpha_1 : (\neg(\alpha_0 = 0 \wedge \alpha_1 = 1)) \rightarrow fail \rightarrow Stop \end{aligned}$$

In (CAVALCANTI; GAUDEL, 2011) it is proved that $Exhaust_T(SP)$ corresponds to all the tests that are valid instances, i.e., that satisfy the constraints, of some symbolic test in $SExhaust_T(SP)$.

The *conf* relation captures reduction of deadlock. Given SP_1 and SP_2 , we have that $SP_1 \text{ conf } SP_2$ if, and only if, whenever SP_2 engages in a sequence of events, that is, a trace that can be accepted by SP_1 as well, then SP_2 can only deadlock if SP_1 may as well. Formally, *conf* can be defined as follows:

$$SP_2 \text{ conf } SP_1 \triangleq \forall t : \text{traces}(SP_1) \cap \text{traces}(SP_2) \bullet \text{Ref}(SP_2, t) \subseteq \text{Ref}(SP_1, t) \\ \text{where } \text{Ref}(SP, t) \triangleq \{ X \mid (t, X) \in \text{failures}(SP) \}$$

For a trace t of a process P and a subset $X = \{a_1, \dots, a_n\}$ of the set of events of P , noted αP , the pair (t, X) belongs to $\text{failures}(P)$ if, and only if, after performing t , P may refuse all events of X . In other words, the parallel composition below may deadlock just after t . We use $\text{proc}(t)$ to represent a *Circus* process that accepts just the execution of t before finishing; it can be defined using prefixing, for example.

$$P \llbracket \alpha P \rrbracket (\text{proc}(t); (a_1 \rightarrow P_1 \square \dots \square a_n \rightarrow P_n))$$

$P \llbracket \alpha P \rrbracket Q$ is the parallel composition of the processes P and Q with synchronisation required on all the events of P , that is, the events in the set αP .

Thus, given a system under test SUT and a specification SP , for $SUT \text{ conf } SP$ to hold, the definition requires that, after performing every single common trace, the failures of SUT are failures of SP . Consequently, after a trace t of SP , SUT may refuse all events refused by SP or accept some of them. Testing for *conf* based on the refusals of SP would be, therefore, useless. What must be tested is that, after every trace t of SP , SUT cannot refuse all events in a set X of events such that $(t, X) \notin \text{failures}(SP)$. Such sets of events are called *acceptance sets* of SP after t .

Thus, tests for *conf* are based on traces and acceptance sets. For a finite trace $s = \langle a_1, a_2, \dots, a_n \rangle$ and a(n acceptance) set $X = \{x_1, \dots, x_m\}$ of events, we define the *Circus* test process $T_F(s, X)$ as shown below.

$$T_F(s, X) = \text{inc} \rightarrow a_1 \rightarrow \text{inc} \rightarrow a_2 \rightarrow \text{inc} \dots a_n \rightarrow \text{fail} \\ \rightarrow (x_1 \rightarrow \text{pass} \rightarrow \text{Stop} \square \dots \square x_m \rightarrow \text{pass} \rightarrow \text{Stop})$$

An example of such a test for *Chrono*, based on trace $\langle \text{tick} \rangle$ and on the acceptance set $\{\text{tick}, \text{time}\}$ is:

$$T_F(\langle \text{tick} \rangle, \{\text{tick}, \text{time}\}) = \text{inc} \rightarrow \text{tick} \rightarrow \text{fail} \\ \rightarrow (\text{tick} \rightarrow \text{pass} \rightarrow \text{Stop} \square \text{time} \rightarrow \text{pass} \rightarrow \text{Stop})$$

An exhaustive test set of a specification SP for *conf* is made of all tests formed by considering all traces of SP , and all the acceptance sets after each of them:

$$\{ T_F(t, X) \mid t \in \text{traces}(SP) \wedge (t, X) \notin \text{failures}(SP) \}$$

Actually $Exhaust_{conf}(SP)$ is defined as a subset of the set above where only minimal acceptance sets are considered, since as soon as some event in a set X is accepted after a trace, any set containing X is an acceptance set after this trace. For instance, the test in previous example is not in $Exhaust_{conf}(\text{Chrono})$, but the two tests below are:

$$\begin{aligned} T_F(\langle tick \rangle, \{tick\}) &= inc \rightarrow tick \rightarrow fail \rightarrow (tick \rightarrow pass \rightarrow \text{Stop}) \\ T_F(\langle tick \rangle, \{time\}) &= inc \rightarrow tick \rightarrow fail \rightarrow (time \rightarrow pass \rightarrow \text{Stop}) \end{aligned}$$

The symbolic counterpart of $Exhaust_{conf}(SP)$ is $SExhaust_{conf}(SP)$, defined in (CAVALCANTI; GAUDEL, 2011).

$SExhaust_T(SP)$ and $SExhaust_{conf}(SP)$ provide basis for defining strategies to select tests by defining subsets of $Exhaust_T(SP)$ and $Exhaust_{conf}(SP)$. Justification is given over uniformity or regularity hypotheses (GAUDEL, 1995) and adequate instantiations (CAVALCANTI; GAUDEL, 2011), or coverage criteria of the specification (CAVALCANTI; GAUDEL, 2014). Hence a motivation for dealing with fault-based selection techniques.

2.4.5 Mutation testing

Mutation testing is recognized as one of the most effective fault-detection techniques (PAPADAKIS; MALEVRIS, 2010). The idea underlying this sort of technique is the reproduction of usual mistakes, often committed by programmers when coding (JIA; HARMAN, 2011). Starting from the original code, the application of *mutation operators* deliberately seed valid syntactic changes, creating a set of potentially faulty programs called *mutants*.

By definition, a mutation operator introduces one syntactic change at a time. All the remaining parts are kept untouched and should perform exactly as the original code. According to the experimental principle, as presented by DeMillo, Lipton and Sayward (1978), the test data that distinguishes all programs by a simple error is so sensitive that it also distinguishes more complex errors. Nevertheless, some approaches explores the application of mutation operators over already mutated programs, thus creating higher order mutants (JIA; HARMAN, 2009).

Mutation testing can also be adapted into the realm of specifications: the systematic injection of feasible modeling faults into specifications allows the prediction of potential defective implementations, provided that the faults affect some observable behavior (BLACK; OKUN; YESHA, 2000).

Testing can benefit from mutation in two ways (JIA; HARMAN, 2011): 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. We say that a test case *kills* a mutant when it exposes the seeded fault. The test coverage analysis in mutation testing context is referred to as the *mutation score*, and it is measured by the ratio of the number of detected faults over the total number of the seeded faults. The top score for mutation coverage

analysis is 1, indicating that the test set in use is sufficient to reveal 100 percent of the artificially introduced faults.

Testing by seeding all possible combinations of syntactic changes in a program is unfeasible, as it will lead to the generation of an impracticable number of mutants (JIA; HARMAN, 2011). Considering this fact, it is important to plan mutation testing activities targeting specific kind of faults or fault classes and, therefore, it is necessary to choose some adequate set of mutation operators in order to induce these faults. Given that the seeded faults lead to variations of the source that are not far from the correct one, mutation testing promises effectiveness in identifying adequate test data that can be used to find actual faults (GEIST; OFFUTT; HARRIS, 1992).

There are two fundamental hypotheses that are taken into consideration when planning mutation test activities: the coupling effect and the competent programmer hypothesis (DEMILLO; LIPTON; SAYWARD, 1978). In a nutshell, they justify the reasons for keeping the mutated versions close to the original source by stating that simple errors scale to huge consequences and assuming that a programmer is prone to write correct code. These hypotheses are discussed in the next subsection, along with the introduction of mutation operators.

2.4.5.1 Mutation operators

A mutation operator is a function that takes as input an original piece of code and generates as outputs its mutated versions, inserting some modification that potentially affects the behaviour of the program under the semantics of the language. More than one mutant may be produced by a mutant operator when the input contains multiple occurrences of code statements within the domain of the operator, e.g., an operator that negates boolean variables will produce a mutant for each boolean variable present in the code. In the case of the nonexistence of statements within the domain of the operator, no mutants are generated.

As stated, the mutants should be constructed by introducing small syntactic changes. In order to design plausible operators, useful for fault-based testing, DeMillo, Lipton and Sayward (1978) proposed the competent programmer hypothesis: mistakes unintentionally introduced by real programmers tend to yield close to correct results. Keeping the hypothesis in mind, the artificially seeded faults should be realistic in relation to the ones that are expected to occur naturally. Several works have proposed mutation operators that attempt to mimic the sort of faults caused by inattentive developers. As noted by Agrawal *et al.* (1989), we list three of the classic programming mistakes:

Incorrect variable name: the replacement of the name of a variable with the name of some other that is, as well, part of the context;

Wrong relational operator: the use of a relational operator that does not yield the intended result for an expression, e.g., using $\leq$ instead of $<$ when iterating over zero-indexed

sequences, such as arrays in most programming languages;

Statement misplacement: placing the statements in the wrong order, causing unexpected function returns or affecting the result of a sequence of dependent expressions.

[Basili and Perricone \(1984\)](#) presented a well structured categorization for software development mistakes, divided in two top classes: omission and commission. Omission mistakes occur when a programmer forgets to include a statement or part of it, and commission comprehends the mistakes caused by writing wrong statements. Mutant operators should cover both branches while following the general guidelines:

Single-step changes: mutation operators should cause only minor changes, affecting exclusively a single syntactic element at a time;

Syntactically correct mutants: the resulting code from a mutation operator should be syntactically valid, allowing it to be parsed;

Organized and classified: mutation operators must be designed and organized in classes, allowing the tester to select and apply them based on what is needed when testing for a given fault class.

As a simple example to illustrate how it works, we introduce an arbitrary basic mutation operator *LTtoGE*: it replaces the relational operator $<$ with $\geq$. The operator is applied to the piece of code of the function *max*, shown in [Source code 1](#), which takes as inputs three integer values and outputs the greatest among them. Since there are three occurrences of the $>$ operator in *max*, three mutants are generated by *LTtoGE*, as they are presented in [Source code 2](#), [Source code 3](#) and [Source code 4](#).

Source code 1 – *max* function

```
1: int max(int a, int b, int c){
2:     if(a>b and a>c)
3:         return a;
4:     else if (b>c)
5:         return b;
6:     else
7:         return c;
8: }
```

Source code 2 – *maxm1* mutant

```

1: int max(a,b,c){
2:   if(a<=b and a>c)
3:     return a;
4:   else if (b>c)
5:     return b;
6:   else
7:     return c;
8: }
```

Source code 3 – *maxm2* mutant

```

1: int max(a,b,c){
2:   if(a>b and a<=c)
3:     return a;
4:   else if (b>c)
5:     return b;
6:   else
7:     return c;
8: }
```

Source code 4 – *maxm3* mutant

```

1: int max(a,b,c){
2:   if(a>b and a>c)
3:     return a;
4:   else if (b<=c)
5:     return b;
6:   else
7:     return c;
8: }
```

In [Table 1](#), we show the mutation score analysis for three sets of manually written test cases, comparing the output of the original *max* function with the outputs produced by each of the three *LTtoGE* mutants. Values marked by an underline are discrepant outputs produced by the mutant when tested with the given inputs. Respectively, the first, second and third sets achieved mutation scores of 0.33, 0.66 and 1.

2.4.5.2 Equivalent mutants

The inclusion of a syntactic modification does not necessarily yields a faulty program: such changes may not affect how a program interacts with the outside context or even change components of its internal state. Mutants whose behaviour cannot be distinguished from the original program are called *equivalent mutants* and there are no automated methods to identify their occurrences, since program equivalence is a well known undecidable problem ([BUDD; ANGLUIN, 1982](#)).

For a basic example of how an equivalent mutant may be constructed, given a program *p* and a mutant *m* derived from *p*, *m* is an equivalent mutant if *m* is syntactically different from *p*, but both yield identical observable outputs. This is the case shown in [Source code 5](#) and [Source](#)

Table 1 – Test case sets and their mutation score for [Source code 1](#)

	Test case			Output				Score
	<i>a</i>	<i>b</i>	<i>c</i>	<i>max</i>	<i>maxm1</i>	<i>maxm2</i>	<i>maxm3</i>	
Set 1	1	2	2	2	2	2	2	1/3 killed
	2	1	2	2	2	2	<u>1</u>	
	2	2	1	2	2	2	<u>1</u>	
Set 2	1	1	2	2	2	2	<u>1</u>	2/3 killed
	1	2	1	2	2	2	<u>1</u>	
	2	1	1	2	2	<u>1</u>	2	
Set 3	1	2	3	3	3	3	<u>2</u>	3/3 killed
	1	3	2	3	3	3	<u>2</u>	
	2	1	3	3	3	<u>2</u>	<u>1</u>	
	2	3	1	3	<u>2</u>	3	<u>1</u>	
	3	1	2	3	<u>2</u>	<u>2</u>	3	
	3	2	1	3	<u>2</u>	<u>2</u>	3	

Source: Research data.

[code 6](#), where a mutation operator generated m from p by changing the relational operator $<$ to $!=$. Despite the syntactic change, the output expected from both implementations would result in printing the same sequence of integers, from 0 to 9.

Source code 5 – p program

```
1: for(i=0;i<10;i++)
2:     print(i);
```

Source code 6 – m mutant

```
1: for(i=0;i!=10;i++)
2:     print(i);
```

The analysis of the mutation score is only effective when non-equivalent mutants are considered: if there are no observable differences in the mutant, no test case would be able to reveal a fault. Although known as an undecidable problem ([BUDD; ANGLUIN, 1982](#)), means to identify and avoid equivalent mutants has been subject of extensive research. Approaches based on compiler optimization ([BALDWIN; SAYWARD, 1979](#)), constraint solving ([OFFUTT; PAN, 1996](#)) and program slicing ([HARMAN; HIERONS; DANICIC, 2001](#)) are among many other techniques that may significantly reduce their occurrence.

2.4.5.3 Killing mutants

[DeMillo and Offutt \(1991\)](#) stated that, in order to reveal a seeded fault, three conditions must be observed:

Reachability: the reachability condition requires that the mutated part must be exercised by the test data. As only one change is introduced in each mutant, the execution paths must

follow the same branches for both the original and the mutated versions. If a test does not execute the mutated statement, there is no chance of revealing the seeded fault.

Necessity: the necessity condition requires the perception of a discrepancy between the states of the original and mutated versions. If the state of the mutated program remains identical to the original after exercising the mutated statement, it is guaranteed that all remaining computations will also yield the same outcome as the original version.

Sufficiency: the sufficiency condition states the need of propagating the discrepancy towards an observable output. If the affected state value remains only internal and have no effect over the future observable manifestations, it will be impossible to differentiate the mutant from the original program via testing.

Testing approaches that take all three conditions into consideration are classified as *strong mutation*. In short, strong mutation requires the mutated statement to be exercised, that such execution yields a discrepancy in the program state and that the discrepancy reflects, at some point, in a distinct observable output. It is not trivial to conduct a test that matches such requirements and some weaker techniques proposes a trade-off between testing effort and effectiveness (JIA; HARMAN, 2011).

Howden (1982) introduced a testing approach named as *weak mutation*, presenting a technique that satisfies the reachability and necessity conditions. The authors divided classes of programming statements in what they called “program components”: variable reference, variable assignment, arithmetic expression, relational expression and boolean expression. By targeting each component using a specific test case generation strategy, they assured that the exercised statement would cause a discrepancy in the program state. The literature uses the name weak mutation as a classification for techniques that satisfy some of the conditions.

Weak mutation does not require a complete execution process: once a mutated statement is reached, the mutant is considered killed. It might even be possible to generate test cases without building the mutants themselves: the test data constraints can be determined beforehand (WOODWARD, 1990). On the other hand, a different outcome of an isolated component might not affect the overall program behaviour, sacrificing test effectiveness: the aforementioned trade-off situation.

Woodward and Halewood (1988) argue that strong and weak mutation are extreme ends, introducing the *firm mutation* concept, looking for a middle ground between these two ends. The authors relate the introduced changes with a time line for the program execution in a way that the changed part might persist for a limited number of cycles. The result is an approach for performing mutation testing on partial executions of program fragments in a highly interactive development environment.

2.5 Final considerations

This chapter assembled all the background information and techniques that should be introduced before the presentation of the contributions from this work: formal methods in software engineering, specification with process algebra, the *Z* notation, concepts of specification based testing, mutation testing, the *Circus* language semantics and its testing theory.

It is natural that the extension of the covered subjects is not compatible with the limited space available in a single chapter, but efforts were made to select the most relevant points from each topic. All discussions were exposed in a summarized form that should provide enough foundation for keeping up with the remaining of the document. Nevertheless, extensive references are offered in all topics and should be useful for gathering further details.

FORMAL MUTATION TESTING IN *CIRCUS*

3.1 Considerations

This chapter details the contributions for fault based testing in *Circus*, as published in (ALBERTO *et al.*, 2016). In [section 3.2](#), we extend the testing theory to consider mutation testing and describe the approach to generate tests based on mutants. The mutation operators used to generate the mutants themselves are defined and analyzed in [section 3.3](#). In [section 3.4](#), we give an example of how some mutation operators would affect a concurrent specification.

3.2 Mutation testing in *Circus*

We assume that there is a specification (model), which is a *Circus* process that describes what the implementation should do. A *mutant* is also a *Circus* process, somehow related to the original specification; it represents a fault in that specification. In what follows, we may also refer to mutants as *faulty models*. In general, given a mutant defined as a *Circus* process, it can be used to generate tests to identify the fault it represents, that is, to “kill the mutant”. We apply the approach in (AICHERNIG; JIFENG, 2008) for mutation testing based on refinement, but consider the particular case of state-rich process algebraic models, and *Circus* in particular.

A mutant *FM* is *of interest* if there exists at least one test that can kill it. This is not the case if it is a refinement of the specification. This is the counterpart at the model level of the well known problem of equivalent mutants at the program level. In such a case, the mutant is not a faulty model and so, not relevant.

In [subsection 3.2.1](#) we formalize mutation testing when traces refinement is the conformance relation of interest. [subsection 3.2.2](#) presents an example: a mutant and the tests to kill it. In [subsection 3.2.3](#) we consider mutation when testing against the *conf* conformance relation. Finally, in [subsection 3.2.4](#), we introduce another form of traces, closer to the text of

the specification, that can be used to relate tests and mutation points in the specification.

3.2.1 Mutation against traces refinement

In the context of traces refinement, we consider a faulty model FM to be of interest if it is not a traces refinement of the specification model SP . In this case, there is at least one trace of FM that is not a trace of SP ; this is not the empty trace $\langle \rangle$, because $\langle \rangle$ is a trace of all processes. To detect the fault in FM , we can use a test $T_T(s, a)$ characterised by any of the minimal traces $s \frown \langle a \rangle$ of FM that are not traces of S . Formally, we define the set $FBTests_T^{SP}(FM)$ of fault-based tests characterised by FM with respect to a specification SP as shown below.

Definition 1.

$$FBTests_T^{SP}(FM) = \{s : \text{traces}(SP); a : \Sigma \mid s \frown \langle a \rangle \in \text{traces}(FM) \setminus \text{traces}(SP) \bullet T_T(s, a)\}$$

This is the set of all tests $T_T(s, a)$, formed from traces s of SP and events a from Σ such that $s \frown \langle a \rangle$ is a trace of FM , but not of SP . A mutant FM is killed by any test from $FBTests_T^{SP}(FM)$. As a direct consequence of its definition, we have that $FBTests_T^{SP}(FM)$ is a subset of $Exhaust_T(SP)$, since it contains tests $T_T(s, a)$, where $s \in \text{traces}(SP)$ and $s \frown \langle a \rangle \notin \text{traces}(SP)$.

Before generating tests based on a mutant FM , a first step is the confirmation that it is not a traces refinement of the model SP . For that, it is of value to use a refinement model checker, like FDR (GIBSON-ROBINSON *et al.*, 2014) for CSP, for example, to check $SP \sqsubseteq_T FM$. If this does not hold, FDR provides a counterexample: a minimal trace of FM that is not a trace of SP . As said above, this identifies a test to detect the fault specified in FM .

3.2.2 Circus mutation example

Figure 6 presents the mutant *MutatedChrono* of the *Chrono* process, which is obtained by the introduction of the negation ($\neg$) operator in the first guard of the action *Run*. The following change in behavior arises from this mutation: like *Chrono*, *MutatedChrono* starts with the *AInit* operation that initialises *min* and *sec* to 0 and, after a *tick*, *IncSec* increases the value of *sec* to 1; afterwards, however, the mutant behaves nondeterministically, either like *Chrono*, that is, executing **Skip** and then *Run* again, or performing *IncMin* and then *Run* again, like *Chrono* in the case *sec* = 0. This erroneously leads to a state where *min* = 1 and *sec* = 1. This mutated state can be later observed via an output on the channel *out* following a *time* event.

Thus $\langle \text{tick}, \text{time}, \text{out}!(1, 1) \rangle$ is a trace of *MutatedChrono*, but not of *Chrono*, for which the only accepted event after $\langle \text{tick}, \text{time} \rangle$ is $\text{out}!(0, 1)$. As seen above, such traces define tests that kill the mutant. An example is the test below:

$$\text{inc} \rightarrow \text{tick} \rightarrow \text{inc} \rightarrow \text{time} \rightarrow \text{pass} \rightarrow \text{out}!(1, 1) \rightarrow \text{fail} \rightarrow \text{Stop}$$

Figure 6 – A mutant of the chronometer

```

process MutatedChrono  $\hat{=}$ 
...
  Run  $\hat{=}$  (tick  $\rightarrow$  IncSec; ( $\neg$  (sec = 0) & IncMin)
            $\square$  ((sec  $\neq$  0) & Skip)))
            $\square$  (time  $\rightarrow$  out !(min, sec)  $\rightarrow$  Skip)
  • (AINit; ( $\mu$  X • (Run; X)))
...

```

Source: Elaborated by the author.

It is a member of $FBTests_T^{Chrono}(MutatedChrono)$.

Actually, this test *may* kill the mutant: the behaviour is nondeterministic due to the overlap of the guards in the choice operator. A complete distinction between “may kill” and “must kill” is given by [Huo and Petrenko \(2009\)](#). Nevertheless, we might state that the test kills the mutant under the complete testing assumption.

3.2.3 Mutation against conf conformance relation

Given a mutant FM , it is of interest if $\neg (FM \text{ conf } SP)$. In this case, there is at least one common trace s of SP and FM for which $\neg (Ref(FM, s) \subseteq Ref(SP, s))$. Therefore, according to the definition of $Ref(P, s)$, there is at least one set of events X such that $(s, X) \in failures(FM)$, but $(s, X) \notin failures(SP)$.

The detection of the fault specified in FM with respect to *conf* is based on tests $T_F(s, X)$ characterized by $(s, X) \in failures(FM)$, where $(s, X) \notin failures(SP)$, and s is a trace s of SP and FM . The full set $FBTests_F^{SP}(FM)$ of fault-based tests for *conf* characterized by FM with respect to SP is defined as follows.

Definition 2.

$$\begin{aligned}
 FBTests_F^{SP}(FM) = \\
 \{s : traces(SP) \cap traces(FM); X : \mathbb{P}\Sigma \mid \\
 (s, X) \in failures(FM) \setminus failures(SP) \bullet T_F(s, X)\}
 \end{aligned}$$

As a direct consequence of the above definition, we have that $FBTests_F^{SP}(FM)$ is a subset of $Exhaust_{conf}(SP)$, since it contains tests $T_F(s, X)$, where $s \in traces(SP)$ and $(s, X) \notin failures(SP)$. In words, X is an acceptance set of SP after s .

Theorem 1. For every mutant FM of a specification SP , the following statements are equivalent:

1. FM is of interest; and

2. $FBTests_T^{SP}(FM) \neq \emptyset \vee FBTests_F^{SP}(FM) \neq \emptyset$.

Demonstration. If FM is of interest, it is not a refinement of SP , that is, either it is not a traces refinement of SP , or it does not satisfy $FM \text{ conf } SP$.

In the first case, as shown in (CAVALCANTI; GAUDEL, 2011), there exists some $T \in Exhaust_T(SP)$ such that its execution against FM yields a *fail* verdict. By definition of $Exhaust_T(SP)$, there are s and a , such that $T = T_T(s, a)$ and $s \in traces(SP) \wedge s \frown \langle a \rangle \notin traces(SP)$. From the definition of $T_T(s, a)$, since the *fail* event is reached, $s \frown \langle a \rangle$ is a trace of FM . Thus, from Definition 1, T belongs to $FBTests_T^{SP}(FM)$, and so $FBTests_T^{SP}(FM) \neq \emptyset$.

With a similar argument, the second case, i.e., the FM is not of interest, implies $FBTests_F^{SP}(FM) \neq \emptyset$.

Conversely, if $FBTests_T^{SP}(FM) \neq \emptyset$, there exists some $T = T_T(s, a)$, where $s \in traces(SP)$ and $s \frown \langle a \rangle \in traces(FM) \setminus traces(SP)$. T belongs to $Exhaust_T(SP)$ and by construction yields a *fail* verdict when executed against FM . Therefore, from the exhaustivity result of (CAVALCANTI; GAUDEL, 2011), FM is not a traces refinement of SP .

The proof that $FBTests_F^{SP}(FM) \neq \emptyset$ implies that $\neg FM \text{ conf } SP$ is similar. $\square$

In Theorem 1, we formalize the previously introduced notion of mutants of interest: a mutant is of interest if its fault can be exposed by, at least, a test.

Returning to *MutatedChrono* example, another change of behavior is that it introduces a deadlock. When the mutated external choice is reached in a state where $sec = 0$, because all the possible choices are guarded by the negation of this condition, there is a deadlock. Due to the equation $sec' = (sec + 1) \text{ mod } 60$ in the *IncSec* schema, it occurs after sixty *tick* events. After such a trace, *Chrono* must accept one more *tick* event, or one *time* event, but the mutated specification refuses both. This leads to the following tests, each of them killing the mutant under the complete testing assumption.

$(inc \rightarrow tick)^{60} \rightarrow fail \rightarrow tick \rightarrow pass \rightarrow \text{Stop}$
 $(inc \rightarrow tick)^{60} \rightarrow fail \rightarrow time \rightarrow pass \rightarrow \text{Stop}$

We use $(inc \rightarrow tick)^{60}$ to denote a prefixing action where the events *inc* and *tick* are offered in alternation, starting with *inc*, 60 times.

When generating tests based on given a mutant FM , a first step is the confirmation that it is a traces refinement, but not a failures refinement of SP . If we can use a refinement model checker, we can check $SP \sqsubseteq_T FM$, and then use the counterexample for $SP \sqsubseteq_F FM$. In the case of FDR, for example, the counterexample is a minimal set of acceptances of FM that is not a set of acceptances of SP . Alternatively, it gives a set of refusals of FM that is not a set of refusals of SP . It is this set of refusals that can be directly used to define a test to detect the fault specified in FM .

In our framework, a set of mutants M of SP defines a set

$$T_T^M \triangleq \{M \bullet FBTests_T^{SP}(M)\}$$

of subsets of $Exhaust_T(SP)$ and a set

$$T_F^M \triangleq \{M \bullet FBTests_F^{SP}(FM)\}$$

of subsets of $Exhaust_{conf}(SP)$. Therefore, M is the basis of a test selection method in the sense of (CAVALCANTI; GAUDEL, 2011). Besides, given a test suite T , it is adequate for M if, for each $m \in M$, $T \cap FBTests_T^{SP}(m) \neq \emptyset$ or $T \cap FBTests_F^{SP}(m) \neq \emptyset$. From Lemma 1, we know that for a set M of mutants of interest, there is always an adequate test suite.

3.2.4 Specification traces and mutation points

Mutations are related to the text of a specification. Traces and even constrained symbolic traces, however, are not related to the text of the specification. They record a possible history of interactions, and it may well be the case that, in some specific situations, we can relate interactions to events and communications in the text of specification. On the other hand, but there is no record of guards and data operations that may have been evaluated or executed in the path to that interaction.

As an example, we consider the constrained symbolic trace $cst3$ in Figure 5. Since there is only one communication via *out* in the text of *Chrono*, in this special case, we can relate $out!\alpha_0!\alpha_1$ to the anti-penultimate line of its definition. On the other hand, $cst3$ has no record of *AInit* and *IncSec*, and of the guards $sec = 0$ and $sec \neq 0$, which are considered in the path to that interaction and may be the object of a mutation.

Therefore, we may use *specification traces*, as defined in (CAVALCANTI; GAUDEL, 2010b; CAVALCANTI; GAUDEL, 2014), to build the tests aimed at killing a mutant. In (CAVALCANTI; GAUDEL, 2014), specification traces are used to consider data-flow coverage, which is also based on the text of a specification. While *cstraces* are useful for trace selection based on constraints on the traces, they do not support selection based on the text of the specification, as we explain in the sequel.

In specification traces, labels are pieces of the specification: guards (predicates), communications, data operations (schemas) or simple *Circus* actions. In case there are repetitions of identical text pieces in the *Circus* specification, different occurrences are distinguished in the labels using textual tags. The syntactic category of *Labels* is defined in Figure 7; the sets *Pred*, *Exp*, *CName*, *VName*, and *Schema* are those of the *Circus* predicates, expressions, channel and variable names, and Z schemas (OLIVEIRA, 2006).

In Figure 8, we present two rules of the transition system that characterizes specification traces. These transition rules correspond to those for the operational semantics shown in Figure 4.

Figure 7 – Syntax of specification labels.

$$\begin{aligned}
\text{Label} &::= \text{Pred} \mid \text{Comm} \mid \text{LAct} \\
\text{Comm} &::= \varepsilon \mid \text{CName} \mid \text{CName!Exp} \mid \text{CName?VName} \\
&\quad \mid \text{CName?VName : Pred} \\
\text{LAct} &::= \text{VName}^* : [\text{Pred}, \text{Pred}] \mid \text{Schema} \mid \text{VName} := \text{Exp} \\
&\quad \mid \text{var VName : Exp} \mid \text{var VName} := \text{Exp} \mid \text{end VName}
\end{aligned}$$

Source: Adapted from [Cavalcanti and Gaudel \(2014\)](#).

Figure 8 – Transition rules that define specification traces

$$\begin{aligned}
&\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xRightarrow{d?x} (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \mid \text{let } x \bullet A)} \\
&\frac{c \wedge (s; g)}{(c \mid s \models g \& A) \xRightarrow{g}_P (c \wedge (s; g) \mid s \models A)}
\end{aligned}$$

Source: Adapted from [Cavalcanti and Gaudel \(2014\)](#).

We note that, the same notion of configuration is used and the transitions are the same, except for the labels. For instance, for an input, the label $d?w_0$ in the operational semantics uses a symbolic variable w_0 , while in the specification traces it refers to the variable x used in the specification. Moreover, the transition for a guarded action $g \& A$ is observable and records the guard expression.

In [Figure 9](#) and [Figure 10](#), we list some specification traces for the processes *Chrono* and *MutatedChrono*. The use of text pieces from the specifications in labels allows the identification and selection of traces that reach textual constructs affected by mutation.

In ([CAVALCANTI; GAUDEL, 2014](#)), a subset of specification traces, *sptraces*, is considered for the definition of tests satisfying data-flow coverage criteria, namely, the set of specification traces where the last event is an observable event. Some of the specification traces given in [Figure 9](#) and [Figure 10](#) are not *sptraces*. It is the case of *spect1* and *spect'1*, and of all the traces in *Setspect'5*. The specification traces in this last set, for example, are of great interest since they lead to a deadlock that is not in the original specification, and thus provide bases for obtaining tests of $FBTests_F^{Chrono}(MutatedChrono)$.

Converting a specification trace to a *cstrace* requires the definition of an operational semantics for labels. [Figure 11](#) presents its transition rules for input and guard labels. We refer

Figure 9 – Some specification traces for *Chrono*

$$\begin{aligned}
\text{spect1} &: \langle \text{AInit}, \text{tick}, \text{IncSec}, (\text{sec} \neq 0) \rangle \\
\text{spect2} &: \langle \text{AInit}, \text{time}, \text{out!min!sec} \rangle \\
\text{spect3} &: \langle \text{AInit}, \text{tick}, \text{IncSec}, (\text{sec} \neq 0), \text{time}, \text{out!min!sec} \rangle \\
\text{spect4} &: \langle \text{AInit} \rangle \wedge \langle \text{tick}, \text{IncSec}, (\text{sec} \neq 0) \rangle^{59} \wedge \langle \text{tick}, \text{IncSec}, (\text{sec} = 0), \text{IncMin} \rangle \\
\text{spect5} &: \langle \text{AInit} \rangle \wedge \langle \text{tick}, \text{IncSec}, (\text{sec} \neq 0) \rangle^{59} \wedge \\
&\quad \langle \text{tick}, \text{IncSec}, (\text{sec} = 0), \text{IncMin}, \text{time}, \text{out!min!sec} \rangle
\end{aligned}$$

Source: Elaborated by the author.

Figure 10 – Some specification traces for *MutatedChrono*

$$\begin{aligned}
\text{spect}'1 &: \langle \text{AInit}, \text{tick}, \text{IncSec}, \neg (\text{sec} = 0), \text{IncMin} \rangle \\
\text{spect}'2 &: \langle \text{AInit}, \text{time}, \text{out!min!sec} \rangle \\
\text{spect}'3 &: \langle \text{AInit}, \text{tick}, \text{IncSec}, (\text{sec} \neq 0), \text{time}, \text{out!min!sec} \rangle \\
\text{spect}'4 &: \langle \text{AInit}, \text{tick}, \text{IncSec}, \neg (\text{sec} = 0), \text{IncMin}, \text{time}, \text{out!min!sec} \rangle \\
\text{Setspect}'5 &: \langle \text{AInit}, \text{tick}, \text{IncSec} \rangle \wedge \\
&\quad (\langle \neg (\text{sec} = 0), \text{IncMin}, \text{tick}, \text{IncSec} \rangle \mid \langle (\text{sec} \neq 0), \text{tick}, \text{IncSec} \rangle)^{59} \\
&\quad [\text{deadlock}]
\end{aligned}$$

Source: Elaborated by the author.

to Figure 4 for the corresponding rules of the operational semantics. Like in the operational semantics, the configuration is a triple, but instead of a process or action, there is a label associated with a constraint and a state assignment. Labels with no guard, but with an input or output communication, are handled in the same way as input and output prefixes in the operational semantics. When there is a label (g, e, A) , with a guard that may be different from True, if the guard holds in the current state then there is a transition to a label (e, A) with guard True, or no guard, for short. The transition is unlabelled, like in the operational semantics.

The similarity between the operational semantics of labels and of *Circus* is not surprising. Conversion of specification traces (of labels) to a *cstrace* recovers the operational semantics of the *Circus* texts captured in the specification traces.

In Figure 5 and Figure 12 we give the constrained symbolic traces converted from those specification traces listed in Figure 9 and Figure 10. The trace $\text{cst}'4$ is that used in subsection 3.2.2 as a basis for the first test that kills *MutatedChrono*. The trace $\text{cst}'5$ is the unique translation of all the traces in *Setspect'*5; it is the basis of the two other killer tests given in subsection 3.2.2.

Figure 11 – Operational semantics of labels

$$\begin{array}{c}
\frac{c \wedge (s; g)}{(c \mid s \models (g, e, A)) \xrightarrow{\varepsilon} (c \wedge (s; g) \mid s \models (e, A))} \\
\\
\frac{c \wedge T \neq \emptyset}{(c \mid s \models (d?x : T, A)) \xrightarrow{d?w_0} (c \wedge w_0 \in T) \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)}
\end{array}$$

Source: Elaborated by the author.

Figure 12 – Constrained symbolic traces for *MutatedChrono*

$cst'1 : (\langle tick \rangle, true)$
 $cst'2 : (\langle time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 0)$
 $cst'3 : (\langle tick, time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 1)$
 $cst'4 : (\langle tick, time, out! \alpha_0! \alpha_1 \rangle, \alpha_0 = 1 \wedge \alpha_1 = 1)$
 $cst'5 : (\langle tick \rangle^{60}, true) \quad [deadlock]$

Source: Elaborated by the author.

The conversion procedure of *sptraces* into *cstraces* is given in (CAVALCANTI; GAUDEL, 2014) and trivially generalises to specification traces. As seen above, several specification traces may correspond to the same cstrace. This follows from the fact that cstraces record only observable symbolic events, while specification traces record internal events and may distinguish different ways of enchainning the same observable events.

To summarise, in considering mutation testing in *Circus* we use three kinds of traces: specification traces, cstraces, and standard traces. We perform the selection among specification traces and then generate from those some killer tests belonging to $FBTests_T^{SP}(FM)$ and $FBTests_F^{SP}(FM)$, which are defined as sets of concrete tests based on standard traces. Since the operational semantics defines the traces of a specification as cstraces, there is an intermediate form of the tests based on cstraces, that is, symbolic killer tests, that need to be adequately instantiated into some concrete tests.

We must take into account that a specification trace corresponds to one cstrace, but that several specification traces may correspond to the same cstrace, as seen above, and that a specification trace of *FM* that is not a specification trace of *SP* may lead to a cstrace that is the translation of another specification trace of *SP*. Given a mutant *FM* of *SP*, we define a notion of relevant specification trace of *FM*.

Definition 3. Given a specification *SP* and a faulty model *FM* obtained by mutation of *SP*, a

specification trace of FM is *relevant* if:

- it is not a specification trace of SP and its conversion into a cstrace is not a cstrace of SP ,
or
- one of the instantiations of its conversion into a cstrace is part of some failure of FM that is not a failure of SP .

The first case is the starting point for obtaining some tests in $FBTests_T^{SP}(FM)$ and the second case for some tests in $FBTests_F^{SP}(FM)$.

To determine whether a specification trace of FM is relevant, we need to compare its cstrace to those of SP . We note that in the *Circus* testing theory, we use unique (up to predicate equivalence) symbolic representatives of these traces using a fixed ordered alphabet of symbolic variables. That allows a uniform account of these traces, and simplifies the management of names by avoiding renaming complications.

If there are no relevant specification traces of FM , it means that there exist no test for killing FM , that is, the effect of the mutation is not observable, neither for traces, nor for failures. FM is a refinement of SP and is not a mutant of interest.

The existence of non-relevant mutated specification traces is due to the possibility of specifying systems in an abstract way, with hidden operations on some state: the consequences of a mutation of such operations may not be observable. This phenomenon is not specific to *Circus* and similar issues are likely to occur in model-based mutation testing as soon as there is a significant abstraction gap between the model and the SUT.

3.3 Mutation operators

A mutation operator Op is a function that generates a set of mutants for a given specification SP . In each mutant in $Op(SP)$, one fault is inserted. Such an operator only yields well-typed, syntactically correct, mutants. A mutation operator is *valid* for a specification, if it generates at least one mutant of interest.

In this section, we present a list of mutation operators for *Circus*. Many of the mutation operators of CSP presented by [Srivatanakul et al. \(2003\)](#) are directly applicable to *Circus*. These are discussed in [subsection 3.3.1](#). For the data and state aspects of *Circus*, we can benefit from some mutation operators based on fault classes for predicates, like those presented by [Kuhn \(1999\)](#) and enriched by [Black, Okun and Yesha \(2000\)](#). These are discussed in [subsection 3.3.2](#). We need to take into consideration, however, the specific features and particularities of our target language, *Circus*.

A mutation operator is *ideal* for a specification if it generates only mutants of interest; an ideal operator never leads to a refinement. Ideality is a strong requirement, not always achievable.

None of the presented operators is ideal. In fact, there can be no ideal operator for *Circus*: a single change to a process definition can never be guaranteed to lead to a non-refinement for every process. This can be seen by considering a process Q , an arbitrary mutation operator Op , and a process P defined as $P = Q \sqcap Op(Q)$. In applying Op to P , it is always possible to obtain $P' = Op(Q) \sqcap Op(Q)$. Properties of *Circus* guarantee that $P' = Op(Q)$, which does refine P .

3.3.1 Modification of behavioural operators

Three classes of mutation operators are suggested for CSP in (SRIVATANAKUL *et al.*, 2003): process definition, expression, and parameter modification operators. Each operator from the first two classes is considered in this section, and we introduce some variations better suited to *Circus* models. When adequate, we refer to the rules of the *Circus* operational semantics that formalise the affected behaviors.

Parameters are not as important in *Circus* as they are in CSP. Since *Circus* processes can have a state, parametrisation is typically used to define generic processes. In this case, parameters play the role of global constants in the scope of the process definition. We, therefore, do not consider parameter operators here.

Following the terminology in (SRIVATANAKUL *et al.*, 2003), synchronization events, that is, communications without value passing, are called simply “events”. When values are passed, we refer to the events as “communications”. Table 2 lists the operators presented in this section, identifying their classification, abbreviation, name, source and, when available in section C.1, the associated operational semantics rules. The source column displays CSP for operators originally proposed for CSP, CSP* for operators adapted from CSP to match *Circus* constructs, and *Circus* for operators designed specifically for *Circus*.

We describe the operators as functions from texts (of actions) to texts, and identify conditions in which they can or cannot be applied. A mutated process is obtained by applying one of these functions to one of its actions.

In the sequel, we present mutations of events and communications, choice and concurrency operators, name references, and hiding.

3.3.1.1 Mutations of events

Event Drop: (*ped*) removes one (arbitrary) occurrence of a prefixing event. Such mutation is likely to produce processes that are not a refinement of the original specification in two ways: the removal of observable events from some traces and the possibility of deadlock introduction in parallel compositions. This can be checked in the rules (C.1.4) and (C.1.25) of the *Circus* operational semantics, reproduced in section C.1, where output and parallelism are described. The definition of the *ped* operator is as follows.

$$ped(A[e \rightarrow]) = A[]$$

Table 2 – Mutation operators for *Circus*

Class	Abbreviation	Operator	Source	OPS Rules	
Behavioural Operators	Events	ped	Event Drop	CSP	9, 30
		per	Event Replacement	CSP	9, 30
		pes	Event Swap	CSP	9, 30
		pei	Event Insert	CSP	9, 30
	Operators	pco	Choice Operator	CSP	18 - 24
		ppoParSeq	Parallel Composition	CSP*	14,15,25-30
		ppoSeqPar	Sequential Composition	CSP*	25 - 30
		ppoSeqInt	Interleave	CSP*	25 - 30
		ppoParInt	Interleave	CSP*	25 - 30
		ppoNameSet	Parallel State Writing	Circus	25 - 30
		Communic.	pmr	Message Replacement	CSP
	pcr		Channel Replacement	CSP	10 - 12
	pci		Communication Insert	CSP	10 - 12
	pce		Communication Elimination	CSP	10 - 12
	pcs		Communication Swap	CSP	10 - 12
	Name/Hide	pprAction	Action Name	Circus	31 - 33
		pprSchema	Schema Name	Circus	
		pprHide	Hide Events	Circus	
		pprUnhide	Unhide Events	Circus	
Expression Operators	Logical	eni	Negation Insert	CSP	17
		eniGuard	Guard Negation	CSP*	
		elr	Operator Replacement	CSP	
		eld	Operand Replacement	CSP	
	Arithmetical	ear	Operator Replacement	CSP	
		eur	Unary Insertion	CSP	
		eak	Add k to Operand	CSP	
		esk	Sub k from Operand	CSP	
		ead	Operand Replacement	CSP	
	Rel.	err	Operator Replacement	CSP	

Source: Research data.

Note 1. We use $A[t_1]$ to indicate that in the text of action A there is an occurrence of term t_1 . There may be several such occurrences and we assume that we can distinguish them (via their position in the text, for example). $A[t_1]$ refers to a particular occurrence. A subsequent reference to $A[t_2]$ denotes the action obtained by replacing the occurrence of t_1 originally singled out by t_2 . In particular, $A[]$ as used above denotes the text of A with this occurrence replaced by an empty term.

In using the above operator and others to follow, the terms singled out in the action parameter, like $e \rightarrow$ in $A[e \rightarrow]$ above, need to be chosen.

Figure 13 and Figure 14 show the mutants of the *Chrono* process obtained via the mutations: $ped(Chrono[time \rightarrow])$ and $ped(Chrono[tick \rightarrow])$, which remove the occurrences

Figure 13 – Mutation of *Chrono* after dropping *time*

```

process ChronoEDropTime  $\hat{=}$ 
...
  Run  $\hat{=}$  (tick  $\rightarrow$  IncSec; ((sec = 0) & IncMin)
        □((sec  $\neq$  0) & Skip)))
        □(out!(min, sec)  $\rightarrow$  Skip)
  • (AInit; ( $\mu$  X • (Run; X)))
...

```

Source: Elaborated by the author.

Figure 14 – Mutation of *Chrono* after dropping *tick*

```

process ChronoEDropTick  $\hat{=}$ 
...
  Run  $\hat{=}$  (IncSec; ((sec = 0) & IncMin)
        □((sec  $\neq$  0) & Skip)))
        □(time  $\rightarrow$  out!(min, sec)  $\rightarrow$  Skip)
  • (AInit; ( $\mu$  X • (Run; X)))
...

```

Source: Elaborated by the author.

of the synchronization events *time* and *tick* from the *Run* action. Among the traces of *ChronoEDropTime* that are not traces of *Chrono*, there are $\langle \text{out}!(0,0) \rangle$ and $\langle \text{tick}, \text{out}!(0,1) \rangle$, which lead to the tests below:

```

pass  $\rightarrow$  out!(0,0)  $\rightarrow$  fail  $\rightarrow$  Stop
inc  $\rightarrow$  tick  $\rightarrow$  pass  $\rightarrow$  out!(0,1)  $\rightarrow$  fail  $\rightarrow$  Stop

```

They are members of $FBTests_T^{Chrono}(ChronoEDropTime)$.

ChronoEDropTick is an example of a divergent mutant, since *Run* may recurse indefinitely in an endless series of internal actions *IncSec* and *IncMin*. without ever communicating with the environment. Thus, it is discarded.

Event Replacement: (*per*) replaces an event by another one within the current local scope (action) or the global scope (process). It affects traces in similar ways as the *ped* operator, as described by the same rules of the operational semantics.

$$per(A[e \rightarrow], f) = A[f \rightarrow]$$

Event Swap (*pes*) swaps two consecutive synchronization events in an action definition. If we apply it only to distinct events, this operator is prone to yielding mutants of interest, although this cannot be assured. For example, swapping *a* and *b* in $a \rightarrow b \rightarrow \mathbf{Stop} \sqcap b \rightarrow a \rightarrow \mathbf{Stop}$ leads to a traces refinement. The behavior is defined by the same rules (C.1.4) and (C.1.24) of the *Circus* operational semantics.

$$pes(A[e \rightarrow f \rightarrow]) = A[f \rightarrow e \rightarrow] \text{ where } e \neq f$$

Event Insert: (*pei*) inserts an event by duplication. The mutated action may not be of interest if the duplication occurs inside a loop.

$$pei(A[e \rightarrow]) = A[e \rightarrow e \rightarrow]$$

This concludes our list of event mutation operators.

3.3.1.2 Mutations of choice and concurrency operators

The following operators target choice and concurrency operators.

Choice Operator: (*pco*) replaces the external choice by the internal choice operator. From rules (C.1.11) and (C.1.13) to (C.1.18) of the operational semantics, we observe that such mutation may introduce deadlocks. The traces of the original and mutated actions are the same, so we always have a traces refinement.

$$pco(A[\square]) = A[\sqcap]$$

We note that replacing an internal with an external choice is not of interest, since $A \sqcap B$ is refined

by $A \sqsubseteq B$, whether we consider traces or failures refinement.

Parallelism and sequence: (*ppo*) in (SRIVATANAKUL *et al.*, 2003), this operator can be used to replace an interleaving, a parallel or a sequential composition with each other. For *Circus*, we need to consider manipulation of state values in concurrent actions. We, therefore, distinguish the possible mutations for each type of composition.

Parallel composition of actions requires the explicit specification of which variables are available for writing by each concurrent action. For instance, in the *Circus* parallel composition $A \llbracket NS_a \mid CS \mid NS_b \rrbracket B$ values written by A into the state variables are only kept for the variables with names in the name set NS_a . The same holds for the action B and the name set NS_b . These sets must be disjoint. Any change of value not matching a name in the respective specified set is cancelled after the parallel composition ends (see rules (C.1.20) to (C.1.21) of the operational semantics). There are several interesting possibilities for injecting faults in parallel composition operators. For instance, to empty one or both name sets in a parallel composition is likely to yield unexpected state configurations and is defined by the following operation.

$$ppoNameSet(A[NS]) = A[\{\}]$$

NS refers to a name set in a parallel composition. Removing or inserting arbitrary elements in NS and other variations of this operator might also be of interest, provided the disjointness of the sets on both sides of the composition is preserved.

We define some specific operators for mutating the nature of the compositions. Such mutations are likely to cause substantial observable changes, although there is no guarantee that the result is a mutant of interest. Specific operators for changing parallel to sequential and the inverse are defined as follows.

$$\begin{aligned} ppoParSeq(A[B \llbracket NS_b \mid CS \mid NS_c \rrbracket C]) &= A[B; C] \\ ppoSeqPar(A[;], CS) &= A[\llbracket CS \rrbracket] \end{aligned}$$

The new sequential composition introduced by *ppoParSeq* ignores the extra parameters of the original parallel composition, that is, NS_a , CS , and NS_b , and the new parallel composition introduced by *ppoSeqPar* is designed to synchronize on all channels in a given set CS and has no write access to any state values. We also define operators for mutating any composition into an interleaving.

$$\begin{aligned} ppoParInt(A[\llbracket CS \rrbracket]) &= A[\llbracket \rrbracket] \\ ppoSeqInt(A[;]) &= A[\llbracket \rrbracket] \end{aligned}$$

Other forms of parallelism may be considered as well, but we do not pursue these here. The needed considerations in all cases are similar to those above.

3.3.1.3 Mutations of communications

The following mutation operators target communications, whose behaviours are described in rules (C.1.4) and (C.1.5). For the sake of conciseness, we do not discuss communications with multiple inputs or outputs.

The removal of an input communication can potentially introduce a syntax error, because it implicitly declares a new variable. Except when the input variable is never used, we may end up with references to variables that are not declared. For mutation operators that change or remove input communications, we, therefore, introduce a declaration of the replaced or removed variable. A *Circus* variable declaration introduces the variable in scope initialized with an arbitrary value from the variable domain.

Message Replacement: (*pmr*) changes the name of a variable used in a communication to another name of a variable of the same type, selected from the current scope. As explained above, missing variable names due to the introduced mutation are redeclared.

$$\begin{aligned} pmr(A[c!e[x] \rightarrow], y) &= A[c!e[y] \rightarrow] \\ pmr(A[c?x \rightarrow], y) &= A[\text{var } x : T \bullet c?y \rightarrow] \end{aligned}$$

T is the type of channel c (and, therefore, of the input variable x).

Channel Replacement: (*pcr*) changes the name of a communication channel to another channel name of the same type.

$$\begin{aligned} pcr(A[c!e \rightarrow], d) &= A[d!e \rightarrow] \\ pcr(A[c?x \rightarrow], d) &= A[d?x \rightarrow] \end{aligned}$$

Communication Insert: (*pci*) is similar to the event insert (*pei*) operator, but applied to a communication instead of a synchronization event. It inserts a new communication by duplicating an existing one.

$$\begin{aligned} pci(A[c!e \rightarrow]) &= A[c!e \rightarrow c!e \rightarrow] \\ pci(A[c?x \rightarrow]) &= A[c?x \rightarrow c?x \rightarrow] \end{aligned}$$

Communication Elimination: (*pce*) removes one arbitrary input or output communication from an action definition. As for the message replacement (*pmr*) operator, if the eliminated communication is an input, a variable declaration must be introduced to declare the input variable and avoid syntactic errors.

$$\begin{aligned} pce(A[c?x \rightarrow]) &= A[\text{var } x : T \bullet] \\ pce(A[c!e \rightarrow]) &= A[] \end{aligned}$$

If the mutated action terminates or deadlocks, it has maximal traces. If the eliminated communication contributes only to the last events of maximal traces, its elimination leads to an action that is a traces refinement of the original action.

Communication Swap: (*pcs*) swaps two consecutive communication events, similarly to the event swap (*pes*) operator.

$$\begin{aligned}
 pcs(A[c1!e1 \rightarrow c2!e2 \rightarrow]) &= A[c2!e2 \rightarrow c1!e1 \rightarrow] \\
 pcs(A[c1!e \rightarrow c2?x \rightarrow]) &= A[c2?x \rightarrow c1!e \rightarrow], \text{ provided } x \text{ is not free in } e \\
 pcs(A[c1?x \rightarrow c2?y \rightarrow]) &= A[c2?y \rightarrow c1?x \rightarrow] \\
 pcs(A[c1?x \rightarrow c2!e \rightarrow]) &= A[\text{var } x : T \bullet c2!e \rightarrow c1?x \rightarrow]
 \end{aligned}$$

When swapping *c1* and *c2* where *c1* is used in an input that declares a variable that may be used in an expression communicated by *c2*, this change leads to a syntactic error. To avoid that, a declaration of the input variable is introduced.

3.3.1.4 Mutations of name references and hiding

Name Replacement: (*ppr*) substitutes name references in the right hand side of an action definition by other process names in scope, including *STOP* and *SKIP*. For *Circus*, we expand this operator to include the manipulation of schema names.

$$\begin{aligned}
 pprAction(A[A1], A2) &= A[A2] \\
 pprSchema(A[S1], S2) &= A[S2]
 \end{aligned}$$

We also introduce two new operators for hiding and unhiding events and communications. The impact of such mutations on observable behavior is similar to event and communication insertion or removal. The channel set for the hiding is an arbitrary subset of the channels in scope. The operational semantics rules describing the behavior of the hiding operator are (C.1.26) and (C.1.27).

$$\begin{aligned}
 pprHide(A1[A2], CS) &= A1[A2 \setminus CS] \\
 pprUnhide(A1[A2 \setminus CS]) &= A1[A2]
 \end{aligned}$$

Similar operators for hiding are also useful at the action level.

3.3.2 Expression modification operators

In *Circus* specifications, some logical and data aspects are modelled along with the definitions of interactions via events. For instance, models typically include guard predicates, variable definitions and assignments, and also data operations defined using *Z* schemas. We consider all these forms of data modelling in the design of mutation operators that capture data and logic faults.

Some fault classes for predicates occurring in software specifications have been proposed by Kuhn in (KUHN, 1999). They seem pertinent if we consider plausible modelling mistakes in *Circus*: variable reference, variable negation, expression negation, associative shifting, operator reference, and missing expressions. We also consider the expression operators introduced for CSP in (SRIVATANAKUL *et al.*, 2003), since they cover most of the mentioned fault classes. Finally, to complement the fault-classes coverage, we also introduce some syntactical operators for *Circus* expressions inspired by the work of Black et al. (BLACK; OKUN; YESHA, 2000).

Negation Insertion: (*eni*) inserts the logical negation operator ($\neg$) before a boolean expression. This mutation is specially interesting when it affects guards, so we have a more specific version of this operator targeting the negation of guards only.

$$\begin{aligned} \text{eni}(A[e]) &= A[\neg e] \\ \text{eniGuard}(A[g\&]) &= A[\neg g\&] \end{aligned}$$

where e is a boolean expression in any *Circus* context, an action or a schema. For example, *MutatedChrono* is obtained from *Chrono* via $\text{eniGuard}(\text{Chrono}[(\text{sec} = 0)\&])$.

Logical Operator: (*elr*) exchanges between the logical “and” ($\wedge$) and “or” ($\vee$) operators. Other logical connectors might be considered, although the most used and more subject to modelling mistakes are these mentioned (SRIVATANAKUL *et al.*, 2003).

$$\begin{aligned} \text{elr}(A[\wedge]) &= A[\vee] \\ \text{elr}(A[\vee]) &= A[\wedge] \end{aligned}$$

Logical Operand: (*eld*) introduces mutations by replacing variable and expression logical operands with *true* or *false* constant values.

$$\begin{aligned} \text{eld}(A[b], \text{true}) &= A[\text{true}] \\ \text{eld}(A[b], \text{false}) &= A[\text{false}] \end{aligned}$$

Arithmetic Operator: (*ear*) replaces a basic arithmetic operator with one of the three others. The four operators considered are sum, subtraction, multiplication and division.

$$\text{ear}(A[\text{op}_1], \text{op}_2) = \text{op}_1, \quad \text{where } \text{op}_2 \in \{+, -, *, /\} \wedge \text{op}_1 \neq \text{op}_2 \bullet P[\text{op}_2]$$

Unary Insertion: (*eur*) inserts the minus modifier in front of an operand. The variable e stands for an arithmetic operand in a process P .

$$\text{eur}(A[e]) = A[-e]$$

Add to Operand: (*eak*) increments an arithmetic operand by an integer constant k . The variable v stands for a numeric variable used in an expression in any schema or action.

$$eak(A[v], k) = A[(v + k)]$$

Subtract from Operand: (*esk*) is similar, but subtracts the integer constant k .

$$esk(A[v], k) = A[(v - k)]$$

Arithmetic Operand: (*ead*) arbitrarily replaces a numeric variable used as an arithmetic operand with another, keeping the types compatible. The variables v and u stand for numeric variables of the same type.

$$ead(A[v], u) = A[u]$$

Relational Operator: (*err*) replaces any of the relational operators $<, \leq, >, \geq, =, \neq$ with any of the others from this same set.

$$err(A[op_1], op_2) = op_1, \quad \text{where } op_2 \in \{<, \leq, >, \geq, =, \neq\} \wedge op_1 \neq op_2 \bullet A[op_2]$$

3.4 Emergency Response System specification example

We include an example to illustrate the how mutation test should be applied to *Circus* specifications: the Emergency Response System (ERS), introduced in (ANDREWS *et al.*, 2013b). Targets, i.e., incidents requiring emergency response, are identified by callers, i.e., members of the public, using the ERS and a set of operationally independent subsystems, such as Phone System, Radio System, Call Center, and Emergency Response Unit (ERU). The ERS must ensure that every call should be sent to the correct target. More details about the ERS can be found in (ANDREWS *et al.*, 2013a). It is used in (MOTA *et al.*, 2014) to assess the deadlock detection of a prototype model checker for *Circus*.

The *Circus* specification in Figure 15 models a subset of the ERS, focusing on the behavior of an emergency response unit manager, specified by the process *ERU*, and of a caller, process *InitiateRescueOrFault*. The latter sends rescue service requests to the *ERU* and can trigger a message-drop fault, to be detected and treated by a fault-recovery component, specified by a process *Recovery* omitted here.

All three processes run concurrently and synchronise on the sets of channels indicated in the definition of their parallel composition. In broad terms, the *ERU* process manages the amount of available and allocated response units. *InitiateRescueOrFault* asks for idle units and

Figure 15 – ERS Specification

```

process ERU  $\hat{=}$  begin
  state Control  $==$  [allocated, total_erus :  $\mathbb{N}$ ]
  InitControl  $==$  [Control' | allocated' = 0  $\wedge$  total_erus' = 5]
  AllocateState  $==$  [ $\Delta$ Control | allocated' = allocated + 1]
  Allocate  $\hat{=}$  allocate_idle_eru  $\rightarrow$  AllocateState; Choose
  ServiceState  $==$  [ $\Delta$ Control | allocated' = allocated - 1]
  Service  $\hat{=}$  service_rescue  $\rightarrow$  ServiceState; Choose
  Choose  $\hat{=}$  if(allocated = 0)  $\longrightarrow$  Allocate
    [allocated = total_erus]  $\longrightarrow$  Service
    [allocated > 0  $\wedge$  allocated < total_erus]  $\longrightarrow$ 
      Allocate  $\square$  Service
  fi
  • InitControl; Choose
end

process InitiateRescueOrFault  $\hat{=}$  begin
  CallCentreStart  $\hat{=}$  start_rescue  $\rightarrow$  FindIdleEru
  FindIdleEru  $\hat{=}$  find_idle_erus  $\rightarrow$  (IdleEru  $\square$  (wait  $\rightarrow$  FindIdleEru))
  IdleEru  $\hat{=}$  allocate_idle_eru  $\rightarrow$  send_rescue_info_to_eru  $\rightarrow$  IR1
  IR1  $\hat{=}$  process_message  $\rightarrow$  FAReceiveMessage  $\square$  fault_activation  $\rightarrow$  IR2
  FAReceiveMessage  $\hat{=}$  receive_message  $\rightarrow$  ServiceRescue
  ServiceRescue  $\hat{=}$  service_rescue  $\rightarrow$  CallCentreStart
  IR2  $\hat{=}$  IR2Out  $\square$  error_detection  $\rightarrow$  FARestartRecovery
  IR2Out  $\hat{=}$  drop_message  $\rightarrow$  target_not_attended  $\rightarrow$  CallCentreStart
  FARestartRecovery  $\hat{=}$  start_recovery  $\rightarrow$  end_recovery  $\rightarrow$  ServiceRescue
  • CallCentreStart
end

channelset ERUSignals  $==$  {allocate_idle_eru, service_rescue}
channelset RecoverySignals  $==$  {start_recovery, end_recovery}
process ERSystem  $\hat{=}$ 
  (InitiateRescueOrFault [ERUSignals] ERU) [RecoverySignals] Recovery

```

Source: Elaborated by the author.

Figure 16 – *ppoSeqInt* *ERSystem* mutants

$$\text{process } ppoParIntERSystem1 \hat{=} \\
(\text{InitiateRescueOrFault} \parallel ERU) \parallel \text{RecoverySignals} \parallel \text{Recovery}$$

$$\text{process } ppoParIntERSystem2 \hat{=} \\
(\text{InitiateRescueOrFault} \parallel ERUSignals \parallel ERU) \parallel \text{Recovery}$$

Source: Elaborated by the author.

the *ERU* allocates them accordingly. If a fault is triggered, *Recovery* logs the occurrence and resend the dropped message to the manager.

This example illustrates mutations that may affect concurrency. The operators *ppoParSeq* and *ppoParInt* are applicable in this context and yield interesting results that we discuss in the following sections.

3.4.1 Mutations by *ppoParInt*

The mutation operator *ppoParInt* replaces a parallel composition with an interleaving, causing the concurrent execution to take place without synchronization between the processes. For this example, this operator can be applied in both parallel compositions to yield the mutants shown in Figure 16.

Both mutants can be killed by test cases based on the fact that the minimal acceptance set for the original *ERSystem* after the empty trace contains only the event *start_rescue*. Other events available due to the mutation are forbidden continuations. The test cases $T_T(s, a_1)$ and $T_F(s, X)$ below are based on the empty trace $s = \langle \rangle$, forbidden continuation $a_1 = \text{start_recovery}$, and acceptance set $X = \{\text{start_rescue}\}$.

$$T_T(s, a_1) = \text{pass} \rightarrow \text{start_recovery} \rightarrow \text{fail} \rightarrow \text{Stop}$$

$$T_F(s, X) = \text{fail} \rightarrow \text{start_rescue} \rightarrow \text{pass} \rightarrow \text{Stop}$$

They can both be used to kill *ppoParIntERSystem1* and *ppoParIntERSystem2*.

3.4.2 Mutations by *ppoParSeq*

The parallel composition in the definition of *ERSystem* is changed to a sequential composition by the concurrency mutation operator *ppoParSeq*. The result of the mutation is shown in Figure 17. In the two produced mutants, the first process of the resulting sequential composition has a non-terminating looping behavior: both *ERU* and *InitiateRescueOrFault* are non-terminating. This makes the second process in the sequential composition unreachable. As an strategy to kill the mutants, test cases can be based on traces that exercise events exclusively available in the unreachable process.

Figure 17 – Mutations by *ppoParSeq*

process *parSeqERSystem1* $\hat{=}$
(InitiateRescueOrFault; ERU) [RecoverySignals] Recovery

process *parSeqERSystem2* $\hat{=}$
(InitiateRescueOrFault [ERUSignals] ERU); Recovery

Source: Elaborated by the author.

The behavior of the mutant *parSeqERSystem1*, for example, is restricted to that defined by *InitiateRescueOrFault*, at the left of the sequential composition, in parallel with *Recovery*. Such behavior is identical to that of *ERSystem* while it holds that *allocated* < *total_erus* in the *ERU* process. To expose this, we need a trace to reach a state in *ERSystem* where *allocated* = *total_erus* and check for a forbidden continuation $a_2 = \text{allocate_idle_eru}$ and acceptance set $X_2 = \{\text{wait}\}$. Below, we have a trace s_1 of both *parSeqERSystem1* and *ERSystem*.

$$s_1 = \langle \text{start_rescue}, \text{find_idle_eru}, \\ \text{allocate_idle_eru}, \text{send_rescue_info_to_eru}, \\ \text{fault_activation}, \text{drop_message}, \text{target_not_attended} \rangle$$

In *ERSystem*, in the execution of this trace, the state of the process *ERU* is changed by increasing in the value of *allocated* by 1. Such trace can be repeatedly observed in the execution of *ERSystem* up to five times, until *allocated* = *total_erus*, when no more units are available and the minimal acceptance set is a single *wait* event. The mutant *parSeqERSystem1*, on the other hand, is able to perform the forbidden continuation *allocate_idle_eru* and, therefore, can be killed by the test cases below.

$$T_T(s_1^5 \frown \langle \text{start_rescue}, \text{find_idle_eru} \rangle, \text{allocate_idle_eru}) \\ T_F(s_1^5 \frown \langle \text{start_rescue}, \text{find_idle_eru} \rangle, \{\text{wait}\})$$

We use s_1^5 to represent the trace containing five consecutive copies of s_1 . We omit the explicit definition of the above tests due to their size.

The mutation inflicted in *parSeqERSystem2* removes the *Recovery* process from the parallel execution, as it becomes the second part of the sequential composition. So, the events in this process are absent in the mutant. The following trace

$$s_2 = \langle \text{start_rescue}, \text{find_idle_eru}, \text{allocate_idle_eru}, \text{send_rescue_info_to_eru}, \\ \text{fault_activation}, \text{error_detection}, \text{start_recovery} \rangle$$

is both a trace of *parSeqERSystem2* and *ERSystem*. In *ERSystem*, however, the next events are from *Recovery*, which is not reachable in the mutant *parSeqERSystem2*. So, the following tests

can kill the mutant.

$$T_T(s_2, end_recovery) =$$

$$\begin{aligned}
& inc \rightarrow start_rescue \rightarrow pass \rightarrow find_idle_eru \\
& \rightarrow pass \rightarrow allocate_idle_eru \rightarrow pass \rightarrow send_rescue_info_to_eru \\
& \rightarrow pass \rightarrow fault_activation \rightarrow pass \rightarrow error_detection \\
& \rightarrow pass \rightarrow start_recovery \rightarrow pass \rightarrow end_recovery \\
& \rightarrow fail \rightarrow \mathbf{Stop}
\end{aligned}$$

$$T_F(s_2, \{log_fault\}) =$$

$$\begin{aligned}
& inc \rightarrow start_rescue \rightarrow fail \rightarrow find_idle_eru \\
& \rightarrow fail \rightarrow allocate_idle_eru \rightarrow fail \rightarrow send_rescue_info_to_eru \\
& \rightarrow fail \rightarrow fault_activation \rightarrow fail \rightarrow error_detection \\
& \rightarrow fail \rightarrow start_recovery \rightarrow fail \rightarrow log_fault \\
& \rightarrow pass \rightarrow \mathbf{Stop}
\end{aligned}$$

Considering the trace s_2 , we can use the forbidden continuation $end_recovery$ and the minimal acceptance set $\{log_fault\}$ to obtain the tests shown above.

3.5 Final considerations

In this chapter, we have formally defined mutation testing for *Circus* by characterizing the exhaustive sets of tests that can kill a given mutant, considering both traces refinement and *conf*, that is, process refinement in *Circus* as a whole. We also present a technique to select tests from these sets based on specification traces of the mutants. An extensive collection of operators to generate mutants considering faults related to both reactive and data manipulation behaviour was introduced as well.

TOOL DEVELOPMENT

4.1 Considerations

Automating procedures has been a major concern for software testing. The applicability of all methods for assessing software quality is affected by complications when it comes to tool development ([PAPADAKIS; MALEVRIS; KALLIA, 2010](#)). Such statement is also valid for mutation testing, comprehending a group of high cost methods that are uncomfortable to automate.

Nevertheless, in this chapter, we introduce prototypical tools that aims at helping in experimenting with the mutation testing theory that we established for *Circus* specifications. In [section 4.2](#), we describe the construction of a set of tools to mutate a specification, then analyze and slice the mutants in order to generate traces that reach the seeded change. Such procedures targets the selection of test cases that have potential to reveal the introduced changes.

In order to enumerate and experiment with traces in *Circus* and given the unavailability of similar tools, we have built a symbolic evaluator based in the formal operational semantics of the specification language. The complete elaboration process is described in [section 4.3](#).

4.2 Mutation testing automation

In a coarse manner, performing mutation test might be divided into three top level tasks: generating mutants from the original program, selecting test cases to kill the mutants and applying the tests to check implementations. The first two activities are not trivial to automate and, despite the effort in tool development, the result still requires a substantial amount of interaction with the tester.

[Budd \(1980\)](#) has found that the number of mutants generated for a program is related to the product of the count of data references times the count of data objects. The estimate was

validated empirically over several sample programs (OFFUTT *et al.*, 1996). The conclusion indicates that it is necessary to handle a large number of mutants, even for average-sized programs. Since it is often unfeasible to indiscriminately apply all available mutation operators, a tester should begin by manually selecting which ones are relevant to the fault classes of interest, in order to control the cost-effectiveness relation in a mutation testing activity.

Program Testing Using Mutants (ProTeUM) (DELAMARO; MALDONADO, 1996), a relevant suite of tools for mutation testing, has been presented in 1996. Initially, it was targeted for unit testing in C language, but further improvements extended its usefulness to other realms, such as interface testing (DELAMARO *et al.*, 2001). As usual with mutation testing automation tools, the development time span covered several years and rarely overcome the practical barriers to reach production level, remaining restricted to academic contexts.

The Mothra/Godzilla software testing environment (DEMILLO *et al.*, 1988), considered the most important contribution in automating mutation testing (PAPADAKIS; MALEVRIS; KALLIA, 2010), was developed over decades before achieving a reasonable level of automation for mutation testing in C and Fortran programming languages and yet requires severe attention from the testing engineer.

With all these difficulties in mind, we have built some prototypical modules to aid the application of the mutation testing theory that was developed in the realm of *Circus* specifications. An overview of how these modules interact to form a tool chain is presented in Figure 18. The first two tasks, i.e., mutant generation and analysis, are presented in the following subsections.

4.2.1 *Circus* mutant generation

To support *Circus* mutation testing and explore the practical aspects of our technique, we have implemented a prototype tool in Python to mutate specifications using the mutation operators presented in section 3.3. The *Circus* notation variant that is accepted as input is consistent with the syntax we introduce in subsection 4.3.2. Nevertheless, the parsing and handling was built in a completely separate python module, using an efficient LALR(1) parser provided by the *Lark*¹ library.

Once parsed, the specification text is represented in a tree structure holding *Circus* constructs in its nodes: the Abstract Syntax Tree, hereby referred to as AST. We show in Figure 19 part of an AST corresponding to the following *Circus* action: $in?x- > out!(x+1)- > Skip$. The labels in intermediate nodes, such as *exp_sum* (arithmetical sum expression), are alias to facilitate the identification of complex syntactic elements.

The mutation operators were implemented using an object-oriented approach, sharing the implementation of common behaviors with super classes and facilitating the future addition of new operators by using independent modules. A simplified hierarchy of mutation operator

¹ <https://github.com/erezsh/lark>

Figure 18 – Overview of the tool chain implementation

Source: Elaborated by the author.

classes is presented in Figure 20. The programming of syntactic operations over the AST follows the guidelines from the visitor design pattern (SCHORDAN, 2006), completely separating object structure and handling algorithms.

All operator classes reference a *CircusContext* object, holding general information about the specification being processed, including the enumeration of all named actions, schemas, channels, variables, events, etc. This object is constructed right after the initial parsing and is reused during the tool life cycle. The collected information is useful when operators need to seed wrong references, e.g., changing a given channel name to another, chosen among the names available in the specification scope.

Each mutation operator produces one copy of the AST for each syntactic change introduced by a first order mutation, i.e., each part of the specification that matches the target syntactic construct is changed and a new modified AST is produced. Considering the piece of code used for the AST example, in Figure 19, if we apply the channel replacement operator, *pcr*, implemented by the class *OPRpcr*, two ASTs will be returned, one for each occurrence of a communication in the original AST. The two changed segments of the tree are shown in evidence in Figure 21.

Following the application of the syntactic changes, the tool traverses the new AST trees and rebuild the *Circus* code, including the added mutation. All mutated specifications are saved in a directory structure that separates the output from each mutation operator. The tool provides

Figure 19 – Part of an AST

Source: Elaborated by the author.

a command line interface that allows the tester to select which operators will be applied over the source code. A sample of the interaction with its command line interface is shown in [Figure 22](#)

Once the mutants are generated, it is necessary to extract information on how to reveal the introduced changes by testing procedures. The approach that we designed for such end is introduced in the next subsection.

4.2.2 Circus mutant analysis

As stated in [subsection 2.4.5](#), a seeded syntactic change is not necessarily propagated to the output and, in such case, cannot be revealed by testing. Moreover, the detection of equivalent mutants is an undecidable problem ([BUDD; ANGLUIN, 1982](#)). In order to maximize the chances of exposing a fault, we look for a way to generate traces that are likely to exercise the piece of code where the syntactic change was introduced: the reachability condition ([DEMILLO; OFFUTT, 1991](#)).

We have elaborated a heuristic approach based on Control Flow Graphs (CFG) ([ALLEN, 1970](#)). A CFG is a directed graph that represents program basic blocks as nodes and control flow

Figure 20 – Class hierarchy for mutation operators

Source: Elaborated by the author.

paths as edges. A basic block is a linear sequence of program instructions. The adaptation of the concept to *Circus* specifications considers forks, including concurrency and choices, as control flow, and parts within these forks as basic blocks. In Figure 27, we present an example of CFGs generated by our tool.

The mutation analysis procedure starts from two *Circus* specification texts: the original version and a mutated version. An overview of the heuristic strategy is presented bellow.

1. **Build Control Flow Graphs:** the CFGs for both original and mutated specifications are built. Original CFG is reused when analyzing multiple mutants.
2. **Find the mutation node:** both graphs are traversed in search for an uncommon node, corresponding to the changed part of the mutated specification. This is the *relevant node*.

Figure 21 – AST changes by *OPR_{pcr}*

Figure 22 – Interaction with the mutant generation utility

```

1 ~/ $python gen_mutants.py spec.circus \
2   --ops OPReniGuard,OPRpcs,OPRpci,OPRppoSeqInt
3 Parsing.. done.
4 Mutant count:
5 OPReniGuard: 0
6 OPRpcs: 1
7 OPRpci: 3
8 OPRppoSeqInt: 2
9 Done.
10
11 ~/ $tree mutants
12 mutants/
13 |--- OPReniGuard
14 |--- OPRpcs
15 |   |--- pcs0.circus
16 |--- OPRpci
17 |   |--- pci0.circus
18 |   |--- pci1.circus
19 |   |--- pci2.circus
20 |--- OPRppoSeqInt
21 |   |--- ppoSeqInt0.circus
22 |   |--- ppoSeqInt1.circus

```

Source: Elaborated by the author.

3. **Find the shortest path:** from the entry point to the relevant node.
4. **Find data dependencies:** check parallel and alternative paths for nodes that might access variables with potential to affect the execution of the shortest path.
5. **Find control dependencies:** check parallel and alternative paths for nodes with potential synchronizations that might be necessary for allowing the specification to progress through the shortest path.
6. **Compose the dependencies:** add all control dependent and data dependent nodes to a final sub graph, along with the path that leads to the mutation.
7. **Constraint check:** to verify if the path is feasible, i.e., to make sure that the constraint yielded by traversing the path can be satisfied by a constraint solver.

In order to illustrate how the heuristic works, we introduce the example in [Figure 23](#), also shown in machine readable version in [Figure 24](#). This *Sample* process forks in four paths, given by one parallel composition and two internal choices among four named actions: *ARead*, that reads the value x from the channel *in*, waits for the *sync* event and assign the value to the state variable v ; *ASync* that just performs the *sync* event; *AShort*, that performs an arbitrary event e ; and *ALong*, that performs the e event five times. The parallel composition between *ASync* and *ARead* performs synchronizations accordingly to the channel set $\{\text{sync}\}$. The process then continues by a sequential composition, joining to a single path where the state variable v is output via the channel *out* inside a guarded action that requires v to be greater than 10.

Figure 23 – Sample specification in *Circus* formal notation

```

channel in, out :  $\mathbb{N}$ 
channel sync, e

process Sample  $\hat{=}$  begin
  state AState  $\hat{=}$  [ $v : \mathbb{N}$ ]
  ARead  $\hat{=}$  in? $X \rightarrow \text{sync} \rightarrow v := x$ 
  ASync  $\hat{=}$  sync  $\rightarrow$  Skip
  AShort  $\hat{=}$  e  $\rightarrow$  Skip
  ALong  $\hat{=}$  e  $\rightarrow e \rightarrow e \rightarrow e \rightarrow e \rightarrow$  Skip
  • (ASync  $\llbracket \{\text{sync}\} \rrbracket$  ARead)  $\sqcap$  AShort  $\sqcap$  ALong;
  ( $v > 10$ ) & out! $v \rightarrow$  Skip
end

```

Source: Elaborated by the author.

Figure 24 – *Sample* specification in machine readable *Circus*

```

channel in, out, sync, b
process Sample ^=
begin
  state AState == [v]
  ARead ^= in?x -> sync -> v := x
  ASync ^= sync -> Skip
  AShort ^= e -> Skip
  ALong ^= e -> e -> e -> e -> e -> Skip
  @ ((ASync [|{|sync|}|] ARead ) |-| AShort |-| ALong);
  (v>10) & out!v -> Skip
end
:run Sample

```

Source: Elaborated by the author.

To simulate the analysis procedure, we seed a mutation from the *eak* operator to the communication that performs the output of *v*, as shown in [Figure 25](#).

Figure 25 – Extract of *Sample eak* mutation

```
(...)
@ ((ASync [|{|sync|}|] ARead) |-| AShort |-| ALong);
(v>10) & out!v+1 -> Skip
(...)
```

Source: Elaborated by the author.

The analysis begin by building the CFGs of both original and mutated specifications, as shown in [Figure 26](#). Both graphs are compared from the entry point in search for changes introduced by the mutation. The entry point of a *Circus* specification CFG is the main or anonymous action of the process under analysis. Once found, the mutation is marked as the relevant node, which will be used as target in the next steps of the analysis.

Figure 26 – CFG comparison for *Sample* and its *eak* mutant

Source: Elaborated by the author.

Figure 27 shows the shortest path sp , highlighted in green, calculated between the entry node (in green) and the relevant node (in red). The mutation will be reached if the path is followed, however, there are dependencies that should be taken into consideration. The next step is to check the path sp for data dependent nodes, i.e., nodes that uses or defines values for variables. The last node in sp , the relevant node itself, represents a prefixed communication of the variable v : nodes that define and/or use variables are made explicit by the label " $[d/u]$ ", where d and u are the sets of defined and used variables, respectively.

We analyze all simple paths, i.e., paths that does not contains repeated nodes, from the entry point to each node in sp that makes use of variables. The procedure in Algorithm 1 reveals all nodes that define values and may affect the results. It should be noticed that the algorithm *simplePaths*, used for finding the simple paths, is a modified depth-first search. Since its complexity conditions the use of the method, it is relevant to state that a single path can be found in $O(V + E)$, being V and E the respective numbers of vertices and edges.

In the example, the search for nodes that define the used variable v revealed two dependencies, as they are highlighted in orange in Figure 27: by traversing backwards, the first defining node found is the assignment of x to v , creating a new dependence on local variable x and triggering a secondary search for nodes that define x , as a consequence, the node that represents the prefixed input of x is also marked as a data dependence.

Algorithm 1 – Finding data dependent nodes

```

1: procedure GETDATADEPNODES(cfg, entry, target)
2:   for  $\forall p \in \text{simplePaths}(\text{cfg}, \text{entry}, \text{target})$  do                                 $\triangleright$  Check all simple paths
3:      $\text{varNames}, \text{depNodes} \leftarrow \emptyset$ 
4:     for  $\forall n \in \text{reverse}(p)$  do                                                     $\triangleright$  Backwards traverse each path
5:        $\text{varNames} \leftarrow \text{varNames} \cup \text{uses}(n)$   $\triangleright$  Includes variables used by node in the set
6:       if  $\text{defines}(n) \cap \text{varNames} \neq \emptyset$  then                                 $\triangleright$  If node defines used vars
7:          $\text{depNodes} \leftarrow \text{depNodes} \cup \{n\}$   $\triangleright$  Add node to dependence set
8:       end if
9:     end for
10:  end for
11:  return  $\text{depNodes}$ 
12: end procedure
  
```

We recalculate the path making sure that all data dependent nodes are included, as highlighted in yellow in Figure 27. The new path contains a segment inside the parallel composition that synchronizes on event *sync*, what may cause the execution to block while waiting for such synchronization. To avoid this situation, the next step searches for control dependent nodes, in order to include such dependencies as well. As shown in Algorithm 2, the path is traversed backwards, and all visible (not hidden) channels are checked. Once a forking node is found, the intersection between the synchronization set and the channels along the path are checked: if one or more of the channels are part of the synchronization set, the alternative branch of the fork is searched for nodes that might exercise such synchronizations.

Figure 27 – CFG for *Sample* example

Source: Elaborated by the author.

The procedure will spot the prefixed synchronization on the event *sync* while traversing the highlighted path. The *fork_10* node, that is part of the path, contains the channel *sync* in its synchronization set, motivating the search for nodes that might perform such event on the

alternative side of the fork. All nodes that synchronize on the channel is then selected as control dependent. In the example, the prefixed communication inside the *ASync* action is marked, as shown in yellow in Figure 27. A sub-graph, presented in Figure 28, is composed using the path that leads to the node of interest, already considering data dependencies, and paths that lead to the new yellow marked edges, which will take care of the control dependencies.

Figure 28 – CFG sub-graph for sliced *Sample* example

Source: Elaborated by the author.

The last step is to check whether such path is feasible. For the example, there is a condition that must be satisfied in order to reach the relevant node: the guarded action requiring $v > 10$. An analysis is performed to gather all nodes that might affect the control flow along the path, and a constraint is built. The constraint is fed as assertions to the theorem prover Z3² (MOURA; BJØRNER, 2008), which is then used to check its satisfiability. For the path of the example, the prover outputs the following model:

Assertions: $[w[1] == w[0], w[1] > 10]$

Dictionary: $\{ 'x': 0, 'v': 1 \}$

Check: z3.SAT

The satisfiable verdict is given by the "Check: z3.SAT" output. The assertions collected along the path consider the input of x , related to the symbolic variable w_0 , the assignment $v := x$, being v related to w_1 and, finally, the requirement imposed by the guarded action, requiring $w_1 > 10$.

An unfeasible path will interrupt the procedure and handle a warning, suggesting that the target node is unreachable. It would be convenient to improve this strategy by searching for longer, yet feasible, paths. This should be considered as a future improvement by the inclusion of a heuristic such as K* (ALJAZZAR; LEUE, 2011), allowing further exploration of alternative paths.

² <https://github.com/Z3Prover/>

Algorithm 2 – Finding control dependent nodes

```

1: procedure GETCONTROLDEPNODES(cfg,path)
2:   rpath  $\leftarrow$  reverse(path) ▷ Reverse the path
3:   depends,hide,unhide,depNodes  $\leftarrow$   $\emptyset$ 
4:   for  $\forall r \in rpath$  do ▷ Backwards traverse
5:     if channels(r)  $\neq \emptyset$  then ▷ Gets the channels used by node
6:       depends  $\leftarrow$  depends  $\cup$  channels(r)
7:     end if
8:     if r is hide then ▷ If the node initiates a hidden action
9:       hide  $\leftarrow$  hide  $\cup$  hideSet(r) ▷ Adds the channel set to the hidden list
10:    end if
11:    if r is unhide then ▷ If the node ends a hidden action
12:      unhide  $\leftarrow$  unhide  $\cup$  hideSet(r) ▷ Adds the channels to the unhidden list
13:    end if
14:    syncItsect = syncSet(r)  $\cap$  depends ▷ Used channels and sync. set
15:    if r is fork and syncItsect  $\neq \emptyset$  then ▷ Node is a fork and sync. channels intersects
16:      hide  $\leftarrow$  h  $\forall h \in$  hide  $\wedge h \notin$  unhide ▷ Channels unhidden along the path
17:      depends  $\leftarrow$  d  $\forall d \in$  depends  $\wedge d \notin$  hide ▷ Hidden channels are not dependencies
18:      for  $\forall p \in$  simplePaths(cfg,r,join(r)) do
19:        depNodes  $\leftarrow$  depNodes  $\cup$  n  $\forall (n \in p)$ 
20:           $\wedge (n \in$  channels(n)  $\cap$  syncItsect)  $\wedge (n \notin$  hide)
21:      end for
22:    end if
23:  end for
24:  return depNodes
25: end procedure

```

Once we have all dependencies enlisted and checked that the path constraint is satisfiable, it is possible to pass the mutant specification through the slicing procedure. All nodes that are not part of the sub-graph are removed from the AST and a reduced specification is built. For the *Sample* CFG example of Figure 27, the sub-graph is a composition of the original shortest path, in green, and the detour motivated by the dependencies, in yellow. The resulting sliced specification is shown in Figure 29, where the named actions that are not used by the final path are removed, and the parallel composition is simplified.

Figure 29 – *Sample* sliced specification

```

channel in, out, sync, e
process SampleS1 ^= begin
  state AState == [ v ]
  ARead ^= in?x->sync->v := x
  ASync ^= sync->Skip
  @
  (ARead [|{sync}|] ASync); (v>10)&out!v+1->Skip
end
:run SampleS1

```

Source: Elaborated by the author.

Since this sliced specification does not contain loops and there is no non-deterministic behaviors, the symbolic execution yields only one solution, shown in Figure 30. As expected, the solution contains the trace that leads the execution to the abnormal state, e.g., it is not in the set of solutions for the original specification, allowing the construction of a test case that kills the mutant. A complete description of the symbolic execution tool is provided in the next subsection.

Figure 30 – Solution output for mutated *Sample* sliced specification

```

Search results:
Solution 1:
ENV  : v |-> V0
CONST: V0 >Int 10
TR   : ( in ? V0 ) ~> sync ~> ( out ! ( V0 +Int 1 ) ) ~> Skip
SP   : ( in ?? x ) ~> sync ~> ( v := V0 ) ~> ( end x )
      ~> ( v > 10 ) ~> ( out !! ( v + 1 ) )

```

Source: Elaborated by the author.

4.3 Symbolic evaluator for *Circus*

The K-Framework is a rewrite-based executable semantic framework for developing tools based on the operational semantics of programming languages (ROSU, 2007).

In this section, we describe how we have defined a configuration structure and a set of rewrite rules within the K-Framework, which, given a *Circus* specification, produces an executable symbolic transition system that mimics the one defined by the formal operational semantics of *Circus*.

The result is a symbolic evaluator that takes the text of a *Circus* specification as input and produces, as output of interest, a constraint and two sets of symbolic traces: one corresponding to the *constrained symbolic traces* of the *Circus* operational semantics (CAVALCANTI; GAUDEL, 2011); the other to the *specification traces*, closer to the specification text, that were introduced in (CAVALCANTI; GAUDEL, 2014) to facilitate the analysis of test coverage criteria.

We have translated each *Circus* operational semantic rule into some equivalent K-Framework rewrite rule or set of rules. The efforts were focused on keeping a close similarity between the rules of the formal semantics and the input format of the K-framework.

Most of the rules were translated in a straightforward way, permitting a crosscheck between the operational semantics and the designed rewriting logic version. Nevertheless, some behaviors, especially those related to the synchronization of nested concurrent processes, required the adoption of more complex strategies. Non-determinism is allowed in *Circus* specifications and it was handled exploiting the backtracking mechanism of the K-framework, guaranteeing the coverage of all possibilities. Next, we present the framework adopted for implementing the tool.

4.3.1 The K-Framework

The K-framework is presented by its authors as a rewrite-based definitional framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined (ROȘU; ȘERBĂNUȚĂ, 2010). It provides many desirable aspects such as modularity, non-determinism and concurrency handling. The framework is presented formally (ROȘU, 2007) and packed in a reference implementation, offering tools to generate executable interpreters that allows state-exploration and reasoning about programs (CIOBACA *et al.*, 2016).

Semantics for programming languages, calculi and other analysis tools are defined in the framework by using a set of rewrite rules and labelled, potentially nested, cell structures, referred to as the *configuration*. The content of the configuration cells keeps relevant information such as the state and general environment for the system/program.

There are two types of rewrite rules: *computational*, which count as computational steps, and *structural* rules, which are used to rearrange the terms so that the computational rules can apply. A *computation* is an element or a sequential list of elements that carries “computational meaning”, that is, a sequence of computational tasks. For instance, the assignment of the value of an expression to a variable is a computation, composed by a sequence of computations which are required to, first, compute the value of the expression and, then, perform the adequate changes to the affected environment configuration cells.

In the K-Framework, computations are syntactical elements of the sort *K* and they are stored and handled inside the configuration cell labelled as *k*. This cell is referred to as the *k cell* or the *computation cell*. Computations have a list structure, capturing the intuition of computation sequentialization, with the symbol $\sim>$ as list the separator, to be read as “followed

by” and the unit symbol “.” (the empty computation).

In particular, computations extend the original language or calculus syntax. When necessary to avoid ambiguities, throughout this report we use the name *k-computation* when referring to a computation inside the k-cell. The k-computations can be handled like any other terms in the rewriting environment, that is, they can be matched, moved from one place to another in the original term, modified, or even deleted.

In such a framework, it is important to distinguish between computations under treatment and computations already completed, for instance, between expressions and their values. To allow this distinction, the framework defines the *KResult* internal syntactic subcategory. Coming back to the variable assignment example, the treatment of the expression will be judged finished when the achieved result is classified under *KResult* category as, for example, an integer value.

The rewrite rules in the K-Framework are unconditional, although they may have ordinary side conditions, and they are context-insensitive, so the rules apply concurrently as soon as they match. They generalize conventional rewrite rules by making explicit which parts of the term is read, write, or ignored.

The framework has been developed with the mechanization of structural operational semantics in mind. The correspondence can be summarized, in a nutshell, as:

- Computation terms and other auxiliary information such as environment and state, are contained in nested *configuration cells*. Computations are kept in special *k-cells*, and other information are organized and managed in cells and sub-cells, depending on the considered language.
- Rewrite rules correspond to transitions of the operational semantics. They describe the evolution of the top configuration cell. Computations progresses are reflected by evolution of k-sub-cells. Changes of environment and state are reflected by changes of other cells.
- Structural rules describe computations rearrangements, so that rewrite rules can match and apply.

Such definitions can be processed by the rewrite engine that is the kernel of the K framework. As say the authors, “interpreters for free” can be obtained from formal language definitions, as well as various tools based on operational semantics. In this implementation, our immediate goal was symbolic traces generation for *Circus*, having in mind a test generation environment as described in (ALBERTO *et al.*, 2016).

Limitations of current implementation

The current version of our symbolic trace generator is limited in dealing with certain aspects of *Circus* specifications:

Type checking is not yet implemented and, as a result, it is necessary to handle each desired type with specific rewrite rules. In the current version, we handle integers and sets, but it is trivial to expand the implementation with rules for more types;

Multiple valued communication is not available, for the moment, we only deal with single valued inputs or outputs;

Schema expressions are not treated in the current version. Besides, the formulas and expressions usable when writing $\mathbb{Z}$ schemas are restricted to the use of native types of the K-framework;

Parallel assignments are not correctly tracked into the specification trace output, but it is done for cstraces;

Specification traces for external choice contains specifications labels of discarded branches. Nevertheless, this undesired behavior is consistent with the operational semantics. It does not affect symbolic traces generation.

Keeping in mind that the rewriting approach of the framework is not usual, even for those who are familiar to software development environments, we introduce the implementation in a more detailed form. The next section presents the design of a simplified machine readable syntax for treating *Circus* specification texts.

4.3.2 Syntax

To establish a simple, text-only, machine-readable syntax for *Circus* specifications, we started from the syntax defined by Feliachi et al. (FELIACHI; GAUDEL; WOLF, 2013) for processing the language in the proof assistant Isabelle/HOL (PAULSON; WENZEL, 2002). We also drawn some inspiration from the machine readable language for CSP, namely CSP_m (SCATTERGOOD; ARMSTRONG, 2011).

We have added some special syntactical constructs for internal use only: they are not expected in the *Circus* specification input by the user, but they are needed as place marks to control the K rewriting engine while performing complex step sequences. When it is the case, such fact will be pointed out in the syntax and semantics descriptions given in the sequel.

The syntax used in our implementation is organized in three modules: the main *Circus* syntax, the syntax for expressions, and a module of useful constructs shared between the other two. We refrain from presenting all modules in this section, focusing only on the elements of the first module.

We present our *Circus* syntax in a BNF style as it is accepted by the K Framework. The root non-terminal element is *CircusSpec* (4.1), a complete *Circus* specification that is a sequence *CircusPars* (4.2) of *Circus* paragraphs.

syntax CircusSpec ::= CircusPars (4.1)

syntax CircusPars ::= CircusPar | CircusPar CircusPars (4.2)

A *Circus* paragraph *CircusPar* (4.3) can represent a process declaration, a $\mathbb{Z}$ paragraph, a channel declaration, a channel set declaration, a name set declaration or the special internal use keyword “:run” (4.6) that allows the user to select which declared process initiates the execution.

syntax CircusPar ::= (4.3)

ProcDecl

| ZParagraph

| "chanset" Id "==" SetExp [strict(2)] (4.4)

| "nameset" Id "==" SetExp [strict(2)] (4.5)

| CompleteCDecl

| ":run" Id (4.6)

The declaration of sets for channels (4.4) and names (4.5) associates an identifier with a set expression *SetExp*. This last element is tagged as *strict*, forcing the K-Framework to advance the computation of *SetExp* before processing the declaration itself.

An element of the sort *SetExp* is defined in the auxiliary syntax module for expressions, which is omitted here. In a brief description, a *SetExp* allows the definition of sets of names using the *Circus* syntax, i.e., comma-separated and delimited by “{” and “}”. It also allows expressions over these sets (such as union, intersection, exclusion, etc.) and identifiers to reference a previously declared named set.

syntax CompleteCDecl ::= "channel" MultiDecl (4.7)

syntax Type ::= "Int" (4.8)

syntax SimpleDecl ::= Id | Id ":" Type (4.9)

syntax MultiDecl ::= List{SimpleDecl, ",", "} (4.10)

Channel declarations are defined by *CompleteCDecl* (4.7). Prefixed by the keyword “channel”, such paragraphs can contain one or multiple (4.10), typed (4.8) or untyped simple declarations *SimpleDecl* (4.9). The channel name is an identifier.

A process declaration *ProcDecl* (4.11) states the association between an identifier, that will reference the process throughout the environment, and a process definition *ProcDef* (4.13) or an action paragraph *Action* (4.20). For specifications that use the second variant, we define a transformation macro rule (4.12) that will convert it into a full process definition and place the given action paragraph as the so-called anonymous action of the process (see (4.13) below).

syntax ProcDecl ::= "process" Id "^=" ProcDef (4.11)
| "process" Id "^=" Action

rule process P:Id ^= A:Action (4.12)
=> process P ^= begin @ A end [macro]

A process definition *ProcDef* (4.13) consists in an optional sequence of process paragraphs *PPars* (4.15) followed by the mandatory anonymous action paragraph *Action* (4.20), after the “@” symbol. The definition is surrounded by the keywords “begin” and “end”.

```
syntax ProcDef ::=                                     (4.13)
  "begin" PPars "@" Action "end"
  | "begin" "@" Action "end"
```

A process paragraph *PPar* (4.14) contains either a $\mathbb{Z}$ paragraph *ZParagraph* (4.16) or a named action declaration.

```
syntax PPar ::= ZParagraph | Id "^=" Action           (4.14)
```

```
syntax PPars ::= PPar | PPar PPars                   (4.15)
```

The *ZParagraph* (4.16) is either a schema or a state declaration. The state declaration is prefixed by the keyword “state” followed by an association between the state name identifier and a single schema definition. The standard non-state schema declaration also allows the association to a *SchemaExp* (4.17).

```
syntax ZParagraph ::= Id "==" SchemaExp               (4.16)
  | "state" Id "==" Schema
```

Although mentioned in this syntax and in *Circus* examples, such as *CashMachine* (CAV-ALCANTI; GAUDEL, 2011), neither the current implementation nor the *Circus* operational semantics offers no support to schema expressions. That explains the reason for the simple definition of *SchemaExp* (4.17). The extension to schema expressions is a subject for future work.

A *Schema* (4.18) has two parts: some variable declarations and a condition, which is a boolean expression. The second part is optional and, when omitted, a structural macro rule (4.19) introduces the “true” keyword at its place.

```
syntax SchemaExp ::= Schema                           (4.17)
```

```
syntax Schema ::= "[" MultiDecl "]"                  (4.18)
  | "[" MultiDecl "|" BoolExp "]"
```

```
rule [ M:MultiDecl ] => [ M | true ] [macro]         (4.19)
```

The most complex non-terminal is *Action* (4.20), which combines CSP syntax with *Circus* elements. It covers CSP constructs like standard process names (4.22), prefixed communication (4.23), guarded actions (4.25), channel hiding (4.28) and several composition operators that are briefly presented by in-line comments. On the *Circus* side there are constructs like $\mathbb{Z}$ schema (4.21), local variable declarations (4.24), recursive actions (4.26) and other *Circus* specific commands (4.29) that are explained in the *Command* (4.31) element. The presence of comparison

```

syntax Action ::= Id                                (4.20)
  | Schema                                          (4.21)
  | "Skip" | "Stop" | "Chaos"                     (4.22)
  | "(" Action ")"
  > Comm "->" Action                               (4.23)
  | "var" Id "@" Action                             (4.24)
  > BoolExp "&" Action                             (4.25)
  > BoolExp "g" Action [strict(1)]
  | "u" Id "@" Action                               (4.26)
  > left:                                          (4.27)
    Action "/" SetExp                               (4.28)
  | Action ";" Action //sequential composition
  | Action "[" Action //external choice
  | Action "|-" Action //internal choice
  | Action "[" SetExp "|" SetExp "|" SetExp "]" Action
  | Action "[" SetExp "]" Action //parallel composition
  | Action "|||" Action //interleave
  > Command                                       (4.29)

```

operators and the keyword “left”, like in (4.27), gives the precedence and the associativity of nested syntactical elements.

The syntax of the communication part of a prefixed action is given by *Comm* (4.30). It can correspond to synchronization events, i.e. channel name identifiers, or single valued inputs and outputs. The use of the strictness attribute in the output is required to force the calculation of the expression value before processing the output construct as a whole. The support for multi-valued inputs and outputs is let to future improvement.

```

syntax Comm ::= Id                                (4.30)
  | Id "?" Id
  | Id "!" Exp [strict(2)]

```

The *Command* non-terminal corresponds to *Circus* commands, such as variable assignment and variable scope finalization, and other special internal use constructs.

```

syntax Command ::=                                (4.31)
  Id "!=" Exp [strict(2)]
  | "end" Id
  | InternalCommand

```

Such internal use constructs, shown in (4.32), are not meant for user input, but used for controlling intermediate steps of the rewriting engine. A complete description of these extra steps is given with the rules for the operational semantics in [section B.2](#)

At last, we need to put all the syntax elements that should be handled as actual values under the framework built-in category *KResult* (4.33). A k-computation is considered completed


```

syntax InternalCommand ::=                                     (4.32)
  | "choice" Set
  | "par" List "," Set "," Set "," Set
  | "hide" Int Set
  | "proc" Int
  | "state" BoolExp [strict]
  | "schema" BoolExp [strict]

```

as soon as the achieved result is of the sort of one of the elements listed under such category.

```

syntax KResult ::= Int | Set | Bool                             (4.33)

```

After parsing a well formed *Circus* specification, the K-Framework will begin the application of semantic rules. For instance, before evaluating the actual behavior of any processes, structural rules are applied to handle the loading of the input specification into the configuration cells. We present in the next section some relevant notions of the K-Framework configuration, along with the configuration structure that we have designed for holding elements of *Circus* specifications.

4.3.3 Configuration

A configuration in the K-Framework is a nested bag of configuration items (or of configuration item terms) referred as configurations cells. During the processing of a *Circus* specification, the configuration provides an abstraction of the state and of the infrastructure needed to process the continuation of the specification, i.e., to compute the subsequent behaviours.

Depending on the considered semantics, configurations may contain cells that keep the current version of: the K-computation, the environment, the store, some analysis results, and various useful bookkeeping information (ROSU, 2007). Given its wide range of use, the K framework does not limit nor impose any restrictions over the design of such a structure, leaving it entirely as a subject of the needs from the semantics being treated and the kind of tool being developed: interpreter, type checker, symbolic evaluator, compiler, trace generator, etc.

There are two kinds of rules describing transitions between configurations: structural and computational rules.

Structural rules process and organize the text of the specification in a way that fits the application of computational rules. They have no counterpart in the operational semantics and can be seen as transformations, preserving the semantics of the text being treated. At initialization, structural rules correctly place the content into the configuration cells, hereby referred to as the *loaded specification*, driving the application of computational rules.

The computational rules correspond to transitions in the operational semantics and describe those configuration changes that represents computations: the reduction steps.

We begin by introducing the configuration structure designed for *Circus* specifications, then we present in [subsection 4.3.3.1](#) the cells that correspond to the loaded specification. These cells memorize the way the specification is stored into various processes and actions. Finally, in [subsection 4.3.3.2](#) we explain the local configurations that deals with the environment of each individual process.

[Figure 31](#) shows an overview of the designed configuration for representing and processing *Circus* specifications. It is presented using the same syntax as accepted by the framework: in a XML style where each tag represents a configuration cell. Such cells can be nested and replicated when explicitly annotated with the attribute “multiplicity”.

Due to the size of the whole structure, the content of some cells is omitted in this overview and presented later.

```
configuration
<top>
  <nextid> 1 </nextid>
  <ch> .Map </ch>
  <nset> .Map </nset>
  <procstrs> ... </procstrs>
  <discarded> .Set </discarded>
  <tempset> .Set </tempset>
  <procs> ... </procs>
  <inits> ... </inits>
  <oks> ... </oks>
  <spectr> .K </spectr>
  <tr> .K </tr>
</top>
```

Figure 31 – Overview of the configuration

The *top* cell encapsulates the whole configuration and contains cells that handle global information, such as: the next fresh integer reference in *nextid*; the map of channel names into channel types in *ch*; the map of set names into the actual sets in *nset*; the structures of loaded processes in *procstrs*; the set of discarded processes references in *discarded*; a temporary set used to convert *Circus* sets into native K-Framework sets in *tempset*; the particular configuration of every process in *procs*; non-resolved initials ready to be synchronized with some other process in *inits*; non-consumed notifications of resolved initials in *oks*; and finally, the results of the treatment of the *Circus* specification, i.e. specification traces and symbolic traces in the *spectr* and *tr* cells.

4.3.3.1 Structures of loaded processes

The content of the cell *procstrs* is a sub structure that represents the processes loaded from the specification text. Presented in [Figure 32](#), the subcell *procstr*, which actually is a bag

of subsubcells, can be replicated as many times as processes are found in the specification text. It contains a name that identifies the process, stored in *pmodel* cell, a map of named action identifiers to action texts in *pact*, a map of schema names to schema texts in *psch*, and the complete process text, which is stored in *pdef*. Whenever a K-computation involves an identifier referring to any of such elements, e.g., named action call, schema or process inclusion, this structure is used to retrieve the required piece of specification text.

Figure 32 – The process textual structures cell *procstrs*

```
<top>...
  <procstrs>
    <procstr multiplicity="*">
      <pmodel> .K </pmodel>
      <pact> .Map </pact>
      <psch> .Map </psch>
      <pdef> .K </pdef>
      <pstate> .Map </pstate>
      <pinv> .K </pinv>
    </procstr>
  </procstrs>
...</top>
```

Source: Elaborated by the author.

4.3.3.2 Local configurations for symbolic evaluation of processes

For processes, sub-processes or actions under symbolic evaluation, we manipulate individual *proc* structures that are subcells of the cell *procs*. This construction is shown in Figure 33. A *proc* cell can be referred as an individual environment for each element that performs K-computations.

This rather complex cell keeps all the elements needed to perform the K-computation that corresponds to the symbolic evaluation of a process or action and how it relates to its context, in the terms of other processes or actions.

The *proc* cells are spawned on demand whenever a new line of K-computation with its own environment is required, e.g., concurrent components in a parallel composition, alternatives of an external choice, inclusion of an independent state-isolated process, etc. These situations are explained along with the description of the corresponding operational semantic rules, in section B.2. The references between these cells reflect the hierarchical tree structure induced by the nested organization of the processes and actions in a *Circus* specification.

A *proc* cell contains: in the *model* subcell, an identifier that references the name of the process which originated it; in *k*, the K-computation cell itself; in *id*, an integer reference that

Figure 33 – The process cell *procs*

```

<top>...
  <procs>
    <proc color="green" multiplicity="*>
      <model> .K </model>
      <k>$PGM:K</k>
      <id>0</id>
      <alt>0</alt>
      <parent>0</parent>
      <env> .Map </env>
      <postenv> .Map </postenv>
      <stack> .List </stack>
      <sync>false</sync>
      <const>true</const>
      <temp> .K </temp>
    </proc>
  </procs>
</top>

```

Source: Elaborated by the author.

is used to globally identify the current environment; in *alt*, a reference to an external choice alternative execution line, if any; in *parent*, a reference to the environment which spawned it; in *env*, a map that associates variable names to symbolic values; in *postenv*, the same sort of map as *env*, but specifically for manipulation of schema post conditions; in *inv*, the state invariant text as written in the process specification; in *stack*, a stack of *env* environments, used for managing nested variable scopes; in *sync*, a boolean flag that is used to put the process evaluation on hold until an adequate synchronization of an initial (see [subsubsection 4.3.3.3](#) and [subsection 4.3.4](#)); in *const*, the constraint on the symbolic values in use by the *env* variables; and an auxiliary cell *temp*, buffering the specification text before outputting it in the generated traces.

4.3.3.3 Handling of initials

The initials cell bag *inits*, shown in [Figure 34](#) is a buffer of observable events or communications that are ready to be performed by an action. Its content is a subset of the set of initials as it is defined in the *Circus* testing theory ([CAVALCANTI; GAUDEL, 2011](#)). The symbolic evaluation of an action can be ready to perform several observable events: it may be the case for actions involving multiple nested parallel compositions or there may be pending synchronizations, or explicitly hidden events or communications. These complications led us to design the configuration structure described below: it makes possible to buffer and manipulate all events or communications that are potentially observable after a given configuration.

One *init* cell holds a reference in *iresp* to the *proc* cell that is currently responsible for its

Figure 34 – The process initial structures cell *inits*

```

<top>...
  <inits>
    <init multiplicity="*">
      <iresp> 0 </iresp>
      <iev> .K </iev>
      <from> .Set </from>
      <ialt> .Set </ialt>
    </init>
  </inits>
...</top>

```

Source: Elaborated by the author.

management. In *iev* is the description of the observable synchronization event or communication that is managed. Moreover, the set *from* holds references to all *proc* cells that are standing for the outcome of such initial, i.e. the cells of the action that originated it and of all other actions that might synchronize when this initial is resolved. Besides, *ialt* is a set of references to alternative *proc* cells, i.e. the ones corresponding to actions that may be discarded as results of external choices when the initial is resolved. We use the term *resolved*, rather than performed, since there are no guarantees that an initial posted during the symbolic evaluation of an action will be actually performed: it might be held indefinitely in a synchronized parallel composition, or hidden by a channel hiding operation.

The resolution of an initial creates an *ok* notification, described in [Figure 35](#). Each *ok* notification contains the event or communication actually performed and the set *to* of the references of all *proc* cells that were standing for the outcome of the resolved *init*.

Figure 35 – The process initial notification structures cell *oks*

```

<top>...
  <oks>
    <ok multiplicity="*">
      <oev> .K </oev>
      <to> .Set </to>
    </ok>
  </oks>
...</top>

```

Source: Elaborated by the author.

The adopted strategy follows a bottom-up approach, in opposition to the top-down one described in the operational semantics. It allows the use of concurrency capabilities as they

were designed: parallel processes in parallel K computation cells, instead of dealing with the semantics of the enclosing compositions. We give more details on the way synchronizations are managed during the symbolic evaluation steps, presented in the next subsection.

4.3.4 Symbolic evaluation of process synchronizations

As mentioned above, the behavior modeling part of *Circus* shares its basis with the CSP process algebra (HOARE, 1985). This long standing and well-known modeling algebra for processes provides a powerful set of constructs to precisely specify parallel behaviors and interactions.

Although allowing the interpretation of parallel computations (threads), as exemplified in (ROSU; SERBANUTA, 2014) and (ŞERBĂNUȚA *et al.*, 2013), the K-Framework does not offer predefined resources for the control of synchronizations between concurrent processes. It is left to the K user dealing with some operational semantics that includes synchronization to design those configurations structures corresponding to a semantically correct parallel behavior.

In a formal presentation of the K framework, Rosu (2007) briefly sketches some examples to introduce the possibility of modeling process algebra semantics: for CCS and π -Calculus. Nevertheless, there is no published operational semantics of CSP in K to the date of the elaboration of this work.

In our context, it was essential to develop in K a correct and efficient semantic representation of the *Circus* concurrent features. That motivated the design of a synchronization strategy that could be embedded in the K-Framework configuration cells, and manipulated respecting its single step rewrite logic rules, while minimizing the inefficiency induced by the need of a global knowledge of the specification to treat occurrences of events, inputs and outputs.

This led us to a bottom-up implementation of the synchronization rules of the OPS, e.g., C.1.20 to C.1.25, to cope efficiently with the conditions on channel sets that occur there, and the possibility of nested synchronizations and hiding.

Our strategy consists in organizing the references between the *proc* cells in a tree structure that splits hierarchically the lines of execution for critical constructions, namely: parallel composition, external choice, channel hiding and independent process inclusion. Given this tree hierarchy, it is possible to manipulate the observable events or communications that may be produced at any depth of the tree and to manage potential synchronizations, deadlocks and hiding.

We show in Figure 36 a graphical representation of the rewriting steps that build the tree structure for a simple example, which contains a parallel composition inside a channel hiding action.

Starting from the text of the main action specification, in the top level *proc* cell, (globally

Figure 36 – Sample graphical representation of a synchronization tree

Source: Elaborated by the author.

identified with the integer reference Id 0), the first step gives a lower hierarchical level to the action subjected to the channel hiding operation. Thus a new *proc* cell, identified with a fresh Id 1, is spawned and noted as a subtree of *proc* Id 0. For convenience, we will refer to a uniquely identified *proc* cell as *proc*#*n*, where *n* is its integer reference. The text of the action of the top-level *proc*#0 is rewritten into an internal syntactical construction that indicates it as a subordinated hidden action, giving the reference to its *proc* cell and the set of hidden channels.

The second rewriting step splits both sides of the parallel composition present in the recently created *proc*#1 into two fresh *proc* cells, *proc*#2 and *proc*#3, respectively for left and right side that are noted as subtrees of *proc*#1. The text of the parallel action inside *proc*#1 is rewritten into an internal syntactical construction that keeps as parameters the references to the new subtree *proc* cells and the set of channels that must be used for synchronizing the corresponding parallel actions.

When a K-computation being processed in a *proc* cell involves an observable event or communication, this item is never immediately added to the traces cells *tr* and *spectr*: it is posted as a potential initial in a new *init* cell as seen in [subsubsection 4.3.3.3](#). In our example, it is the case for *proc*#2 and *proc*#3. Once an initial is posted, the current K-computation remains on hold by lifting the boolean flag inside its *sync* configuration cell. It will remain standing by until a proper *resolution* of the initial is reached.

We say that an initial is *resolved* when it matches the criteria of a resolution rule: some semantic rule that will either suppress the initial (B.42) or write it into the traces (B.54). The resolution causes an *ok* notification inside the *oks* configuration cell. Each notification is

accompanied by a *to* cell, that contains a set of references to all *proc* environments waiting for the initial resolution.

A posted initial will be manipulated and delegated to the upper tree levels by *forwarding rules*. Following the *Circus* operational semantics, there are different sorts of forwarding rules for each splitting construct.

The reason for having this life-cycle for initials, namely: posting, forwarding, resolving and/or discarding, is that the presence of an observable communication/event in a trace depends on the context of the action that triggers it. In two cases, such observable communications/events may never appear in the traces of the main process, i.e., the process under analysis.

First, it may appear in OPS rules where, above the rule, there is a non- ε labeled transition but, under the rule, the transition is labelled with ε (silent). It is the case for the OPS rules C.1.24, C.1.25 and C.1.27, which defines channel hiding and parallel synchronization.

The other case is when an action present in the original configuration of a transition disappears in the target configuration, as in the case of the OPS rules C.1.18 and C.1.19, which defines the behavior for external choice. As a result, observable events/communications triggered by the discarded action will never be written in the traces.

In the following, we introduce the management of initials for the *Circus* constructions mentioned in the two cases above, namely: channel hiding, parallel composition and external choice.

Channel hiding is modeled with rules that may selectively forward initials concerning non hidden channels (B.41) or instantly resolve an initial from the hidden ones (B.42), suppressing it and spawning the respective *ok* notification.

The parallel composition forwarding rules will pass by any initial that is not related to the channels listed in the synchronization set (B.47). Otherwise, the initial will be hold until another initial, matching the synchronization, is produced by the symbolic evaluation of the action at the other side of the composition (B.48).

An external choice operation will indiscriminately forward the initials from both sides, but an alternative reference is marked in the *alt* cell of each, pointing to the *proc* cell in the competing side: the first resolved initial will determine which side is chosen and, consequently, the discarding of all marked alternatives. More details about these rules and the conditions for forwarding initials to upper levels are given in section B.2.

If an initial, after all the manipulations that it may receive in the lower levels, is finally delegated to the *proc*#0 environment, then it is eligible for producing some observable event or communication. However, it is not assured that this will actually occur, since an initial might reach the top-level after the discarding of all those *proc* cells that were matching its outcome. In that case, the initial will be just discarded. Otherwise, the observable event or communication is

appended to the traces and the respective *ok* notification is posted.

A *proc* on hold by the *sync* flag is ready to continue its K-computation once the appropriate *ok* notification exists. The continuation is triggered by a rule (B.55), which tries to match the references of standing-by processes with references in the addressing subcell of *ok* notifications: once matched, a reference is removed from the *ok* notification, the process *sync* flag is released and the standing by K-computation takes place. A synchronized *ok* notification might be addressed to more than one *proc* at the same time and, for that reason, they are not discarded in this step: it would prevent the continuation of the other standing by *proc*. After the removal of all standing-by *proc* references, there is a specific structural rule dedicated to the cleaning of *ok* notifications with empty addressing subcell.

The two last subsections have presented the structure designed to hold the configuration. Refer to [Appendix B](#) or (ALBERTO; GAUDEL, 2017) for a complete description of the structural rules that link the text of *Circus* specifications to this structure, and the rewrite rules that describe the transitions between configurations, according to the *Circus* operational semantics (CAVALCANTI; GAUDEL, 2013).

4.3.5 Conclusion and results

We have established a set of rewrite rules within the K-Framework, which automates the behavior of the transition system defined by the formal operational semantics of *Circus*. We also embedded a subset of the operational semantics of the *Circus* specification traces.

The current result is an executable interpreter, which accepts an input specification text and enumerate its constrained symbolic traces and specification traces. This is an utterly important link in the *Circus* mutation testing tool chain and might be used to other ends as well, given that the rewriting logic framework provides resources to analyze the states and transitions of the simulation.

The specifications we have sampled to experiment with the behaviour of the symbolic evaluator under circumstances of interest are shown in [section 5.4](#). The analysis of the results shows that the execution is faithful to the expected behaviour, accordingly to *Circus* formal operational semantics. Although, a formal proof of correctness is desirable, and is listed as future work.

4.4 Final considerations

We have reported how the development of tools was conducted. Considering the extension of the chapter, details have been abstracted to allow the presentation of the main aspects of each module of the tool chain. Given its complexity and relevance to the project, greater attention was dedicated in the description of the *Circus* symbolic evaluator.

EXPERIMENTS

5.1 Considerations

In this chapter, we describe some experiments that were assembled to validate the concepts and tools developed during the extension of this work. In [section 5.2](#), we report how we selected and used some known *Circus* specifications to check the features of our tool chain. In [section 5.3](#), we present some experiments based on the translation of *Circus* specifications to simpler CSP versions, and used a refinement checker to find mutant killing traces. In [section 5.4](#), we present some sample specifications chosen to check how the symbolic traces generator behaves in comparison with the operational semantics.

5.2 Tool chain validation experiments

We have collected sample *Circus* specifications from the literature, including papers and technical reports, and used them as inputs to our prototype tool chain. The goal is checking whether it was capable of generating mutants accordingly to the presented theory, and produce traces that could be used in building test cases that were likely to kill such mutants.

5.2.1 Methodology

Each *Circus* specification was coded into our machine readable syntax variant, as presented in [subsection 4.3.2](#), and used as input to the tool chain, following the steps bellow:

1. **Mutant generator:** the original specification text is used as input and gets processed by all available operators (see [Table 2](#)). The output is a set of mutant specifications, containing the results of all successfully applied operators.

2. **Mutant analyzer:** the original specification and a mutant are analyzed through their control flow graphs. The tool walks through both graphs in search for differences, allowing the determination of some distinct node or transition that might represent an impact of the mutation. If none is found, the tool suggests the possibility of an equivalent mutant. Otherwise, the spotted node is used as target for the slicing procedure.
3. **Mutant slicer:** a mutant specification, its CFG and the relevant mutation node are used as inputs. This tool will apply heuristics to remove nodes from the CFG that are not relevant in exercising the piece of code where the mutation took place. The output is a sliced specification containing what is needed to reach the mutated part.
4. **Symbolic traces generator:** the sliced *Circus* specification is fed as input and an arbitrary iteration limit is set. The result is a list of symbolic traces along with the execution internal state after the given iteration count or a successful termination.
5. **Output analyzer:** a batch procedure that verifies the output produced by the symbolic traces generator. The result from each sliced mutant is checked against the expected output, accordingly to the result given by the original specification. The comparison takes into consideration the generated traces and the internal state as well.

The verdicts are given in the last step: a mutant is considered killed if the tool finds a symbolic execution configuration where either the trace output or the internal state differs from those yielded by the symbolic execution of the original specification, after the same number of computation steps. The terms *configuration* and *computation* should be considered as in the context of the K-Framework, presented in [subsection 4.3.1](#). For non-deterministic behaviours, a mutant is considered killed if, at least, one of the possible configurations is not part of the set of configurations from the original specification.

5.2.2 Fuel Pump

The fuel pump model ([OLIVEIRA; CAVALCANTI; WOODCOCK, 2009](#)) specifies a process that controls a fuel pump. The system uses seven channels: *init*, *liftNozzle*, *putNozzle*, *pressTrigger*, *releaseTrigger*, *enterAmount* and *reload*. The schema *Init* is used to initialize the pump. Events *liftNozzle* and *putNozzle* represent that the nozzle has been lifted or put back, *pressTrigger* and *releaseTrigger* denote to the pump that the trigger has been pressed or released. *EnterAmount* is used by customers to enter the amount of fuel they want. Finally, *reload* channel reload the pump by a gas station employee.

The channels *enterAmount* and *reload* communicate the amount of fuel that the client wants and the amount of fuel that was loaded into the pump. In the process *Pump*, the schema *PState* contains a natural number *fuelQ* that stores the current quantity of fuel in the pump. The nameless main action of this process is recursive: after a request to initialize, it performs the

initialization of the state, and behaves recursively, executing *PumpIdle*. The state component *fuelQ* is initialized with 5000 by the schema *PInit*. The schema *Reload* adds an input value $q?$ to the current quantity of fuel; *Supply* subtracts the input $q?$ from the current amount of fuel, given that the quantity available is greater than zero.

The specification as seen in (OLIVEIRA; CAVALCANTI; WOODCOCK, 2009) is presented in Figure 37, along with its machine readable translation for use in the tool chain, presented in Figure 38.

Figure 37 – *Pump* specification in *Circus* formal notation

```

channel init, liftNozzle, putNozzle, pressTrigger, releaseTrigger
channel enterAmount, reload :  $\mathbb{N}$ 

process FuelPump  $\hat{=}$  begin
  state PState  $\hat{=}$  [fuelQ :  $\mathbb{N}$ ]
  PInit  $\hat{=}$  [ $\Delta PState$ ;  $q? : \mathbb{N} \mid fuelQ' = 5000$ ]
  Reload  $\hat{=}$  [ $\Delta PState$ ;  $q? : \mathbb{N} \mid fuelQ' = fuelQ + q?$ ]
  Supply  $\hat{=}$  [ $\Delta PState$ ;  $q? : \mathbb{N} \mid fuelQ' = fuelQ \ominus q?$ ]
  PumpIdle  $\hat{=}$   $\left( \begin{array}{l} liftNozzle \rightarrow PumpActive \\ \square reload?q \rightarrow Reload \\ \square init \rightarrow PInit \end{array} \right)$ 
  PumpActive  $\hat{=}$   $\left( \begin{array}{l} putNozzle \rightarrow \mathbf{Skip} \square \\ \left( \begin{array}{l} enterAmount?q \rightarrow pressTrigger \rightarrow Supply; \\ releaseTrigger \rightarrow \mathbf{Skip} \end{array} \right) \end{array} \right)$ 
  •  $init \rightarrow PInit$ ;  $\mu X$  • PumpIdle; X
end

```

Source: Elaborated by the author.

Figure 38 – *Pump* specification in machine readable *Circus*

```

channel init, liftNozzle, putNozzle, pressTrigger, releaseTrigger
channel enterAmount, reload
process Pump ^=
begin
  state PState == [fuelQ]
  PInit == [fuelQ | fuelQ' == 5000]
  Reload == [fuelQ | fuelQ' == fuelQ + q]
  Supply == [fuelQ | fuelQ' == fuelQ - q]
  PumpIdle ^= liftNozzle -> PumpActive []
                (reload?q -> Reload [] init -> PInit)
  PumpActive ^= putNozzle -> Skip []
                (enterAmount?q -> pressTrigger -> Supply;
                 releaseTrigger -> Skip)
  @ init -> PInit; u X @ (PumpIdle; X)
end

:run Pump

```

Source: Elaborated by the author.

5.2.2.1 Results analysis

The mutant generation procedure, configured to apply all available operators, yielded the count as shown in [Table 3](#). After applying the analysis, slicing and symbolic execution steps, the tool chain revealed the number of conflicting configurations, displayed in the last column of the aforementioned table.

It is relevant to break down the results of the mutation analysis. For the sake of space, we have chosen some mutants to show and discuss the outcome from the symbolic execution comparison. For instance, the single mutant generated by the *esk* operator is shown in [Figure 39](#). As designed, the operator subtracts one from each suitable operand in the specification. For *Pump*, the only suitable operand is found on the schema *PInit*, where the value 5000 is assigned to the state variable *fuelQ*. That the new value seeded by the operator is 4999.

Table 3 – Analysis of mutants for *Pump*

Abbreviation	Operator	Mutants	Killed	Total dead configs.
ped	Event Drop	6	3	5
per	Event Replacement	23	18	29
pei	Event Insert	6	5	8
pco	Choice Operator	3	3	7
ppoSeqPar	Sequential Composition	3	2	4
ppoSeqInt	Interleave	3	2	4
pcr	Channel Replacement	12	6	12
pci	Communication Insert	2	1	2
pce	Communication Elimination	2	0	0
pprAction	Action Name	22	17	38
pprSchema	Schema Name	5	2	3
pprHide	Hide Events	6	1	2
eni	Negation Insert	3	1	1
eld	Operand Replacement	6	0	0
ear	Operator Replacement	6	3	3
eur	Unary Insertion	1	1	1
eak	Add k to Operand	1	1	1
esk	Sub k from Operand	1	1	1
err	Operator Replacement	15	10	15

Source: Research data.

Figure 39 – Extract from *Pump esk* mutant

```

(...)
begin
  state PState == [fuelQ]
  PInit == [fuelQ | fuelQ' == 4999]
  Reload == [fuelQ | fuelQ' == fuelQ + q]
(...)

```

Source: Elaborated by the author.

The CFG analysis procedure isolated the node corresponding to the schema call from the process main action, as shown in Figure 40. With the node of interest isolated, the shortest path and the dependencies are calculated to create the sliced mutant specification. The result of the slicing procedure is shown in Figure 41, which contains the necessary steps to reach the syntactic change.

Figure 40 – Extract of the CFG generated by the tool for *Pump esk* mutant

Source: Elaborated by the author.

Figure 41 – *Pump* slice for *esk* mutant

```

process Pump ^= begin
  state PState == [ fuelQ ]
  PInit == [ fuelQ | fuelQ'==4999 ]
  Reload == [fuelQ | fuelQ' == fuelQ + q]
  Supply == [fuelQ | fuelQ' == fuelQ - q]
  PumpIdle ^= liftNozzle -> PumpActive []
              (reload?q -> Reload [] init -> PInit)
  PumpActive ^= putNozzle -> Skip []
                (enterAmount?q -> pressTrigger -> Supply;
                 releaseTrigger -> Skip)
  X ^= PumpIdle; X
  @ init->PInit; X
end

```

Source: Elaborated by the author.

The symbolic execution of the sliced mutant reveals a configuration that did not occur while executing the original specification, shown in [Figure 42](#). It displays a scenario where the mutant has its state variable *fuelQ* assigned to the value 4999, which is unfeasible when considering all the outcomes yielded by the original specification on the same context.

Figure 42 – Symbolic execution comparison for *Pump esk* mutant

```
Running K on sliced spec ../python/cfgmutts/OPResk0_slice.txt ...Done
#####Comparing solutions#####
Solution 2 not in original solution set:
['CONST': 'true',
 'INITS': '',
 'TR': 'init',
 'SP': 'init ~> PInit',
 'ENV': 'fuelQ |-> 4999' ]
#####End solution comparison#####
```

Source: Elaborated by the author.

The symbolic and specification trace that led to the execution of such state are shown in the output as well, respectively *'TR' : 'init'* and *'SP' : 'init > PInit'*. Ruling out non-determinism by the *complete testing assumption* hypothesis (CAVALCANTI; GAUDEL, 2007): there is some known integer n such that, if a test experiment is performed n times, then all possible behaviours are observed, the mutant would be killed by the test case built from the given traces.

5.2.2.2 Considerations

In some cases, a valid syntactic mutant is not necessarily useful when context is considered. One case worth mentioning is the mutant #2, generated by the operator *pprSchema*, shown in Figure 43. The operator replaces schema names by other valid schema names but, in this instance, the replacement caused an error, revealed by the tool while attempting to apply the constraint solving procedure: *OPRpprSchema mutant #2: Constraint error: variable required by schema schema_Reload_13 not available in scope.*

Figure 43 – Extract from *Pump pprSchema* mutant

```
(...)
  Reload == [fuelQ | fuelQ' == fuelQ + q]
(...)
  PumpIdle ^= liftNozzle->PumpActive [] reload?q->Reload [] init->Reload
(...)
```

Source: Elaborated by the author.

The original specification invokes the schema *PInit*, prefixed by the event *init*. In the mutant, the *PInit* schema call is replaced by *Reload*, which makes use of the variable q for

configuring the state. The variable is not available in the given context, incurring in a situation that is not predicted by the semantics. Nevertheless, our tool was capable of detecting and pointed out the case.

5.2.3 Fibonacci Generator

The example *Circus* process in Figure 44, also shown in machine readable version in Figure 45, specifies the behavior of a Fibonacci sequence generator, as introduced in (CAVALLANTI; MOTA; WOODCOCK, 2013). It uses operation schemas to initialize and update state variables.

Figure 44 – Fibonacci specification in *Circus* formal notation

```

channel out :  $\mathbb{N}$ 

process Fib  $\hat{=}$  begin
  (
    FibState  $\hat{=}$   $[x, y : \mathbb{N}]$ 
    InitFibState  $\hat{=}$   $[FibState' \mid x' = y' = 1]$ 
    InitFib  $\hat{=}$  out!1  $\rightarrow$  out!1  $\rightarrow$  InitFibState
    OutFibState  $\hat{=}$   $[\Delta FibState \mid y' = x + y \wedge x' = y]$ 
    OutFib  $\hat{=}$   $\mu X \bullet$  out!(x + y)  $\rightarrow$  OutFibState; X
     $\bullet$  InitFib; OutFib
  )
end

```

Source: Elaborated by the author.

The channel *out* outputs the Fibonacci sequence. The state contains two components: *x* and *y*. The anonymous action at the end of the process is the main action that executes sequentially *InitFib* and *OutFib* actions. *InitFib* outputs 1 twice and calls the state initialization in *InitFibState* schema. In *OutFib*, the next Fibonacci number is calculated and *OutFibState* appropriately updates the state; the number is output and , finally, *OutFib* proceeds recursively.

5.2.3.1 Results analysis

The specification was used as input to the mutant generator procedure, considering all available operators. In Table 4, we shown the results for the operators that succeeded into producing mutants. As in the previous example, the last column displays the number of conflicting configurations from the symbolic execution of the sliced mutant in relation to the configurations of the original specification.

Table 4 – Analysis of mutants for *Fibonacci*

Abbreviation	Operator	Mutants	Killed	Total dead configs.
ppoSeqPar	Sequential Composition	2	0	0
ppoSeqInt	Interleave	2	1	1
pci	Communication Insert	3	3	3
pce	Communication Elimination	3	3	3
pprAction	Action Name	9	5	5
pprSchema	Schema Name	2	1	1
pprHide	Hide Events	3	2	2
eni	Negation Insert	2	0	0
elr	Operator Replacement	2	1	1
eld	Operand Replacement	4	0	0
ear	Operator Replacement	6	3	3
eur	Unary Insertion	3	2	2
eak	Add k to Operand	3	3	3
esk	Sub k from Operand	3	3	3
err	Operator Replacement	20	10	10

Source: Research data.

Figure 45 – *Fibonacci* specification text

```

channel out

process Fibonacci ^=
begin
  state FibState == [x,y]
  InitFibState == [x',y' | x' == y' && x' == 1 ]
  OutFibState == [y',x' | y' == (x + y) && x' == y ]
  InitFib ^= out!1 -> out!1 -> InitFibState
  OutFib ^= out!(x+y) -> OutFibState; OutFib
  @
  InitFib; OutFib
end

```

Source: Elaborated by the author.

The *Fibonacci* process does not contain non deterministic forks and the number of solutions found equals the number of killed mutants. We have chosen one sample from the traces that were selected to generate mutant killing tests for each operator. They are presented in the same form as they are output by the tool chain in [Table 5](#). The first column identifies the operator name and the index of the mutant from which the trace was selected, the second and third columns show the trace and the reached state.

Once again, considering the complete testing assumption hypothesis, the traces and states

Table 5 – Traces and states for *Fibonacci* mutants

Op.#Mutant	Trace	State
ppoSeqInt#1	$out!(V0 + V1)$	$\{y : V0 : Int, x : V1 : Int\}$
pci#0	$(out!1) \sim > (out!1) \sim > (out!1)$	$\{y : V0 : Int, x : V1 : Int\}$
pce#0	$(out!1)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 1\}$
pprAction#3	$out!(V0 + V1)$	$\{y : V0 : Int, x : V1 : Int\}$
pprSchema#0	$(out!1) \sim > (out!1)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1 + 1\}$
pprHide#1	$(.)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 1\}$
elr#0	$(out!1) \sim > (out!1)$	$\{y : V1 : Int, x : V0 : Int, V0 == V1 \vee V0 == 1\}$
ear#3	$(out!1) \sim > (out!1) \sim > (out!0)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 1\}$
eur#1	$(out! - 1)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 1\}$
eak#0	$(out!1) \sim > (out!1)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 2\}$
esk#0	$(out!1) \sim > (out!1)$	$\{y : V0 : Int, x : V1 : Int, V0 == V1, V0 == 0\}$
err#0	$(out!1) \sim > (out!1)$	$\{y : V0 : Int, x : V1 : Int, V1 == 1 \wedge 1 > V0\}$

Source: Research data.

shown in Table 5 are sufficient to build test cases for weak mutation killing, given that they exercise the seeded fault and lead to states that would be unfeasible in the original specification.

5.2.3.2 Considerations

Mutants that have logical expressions replaced with false booleans, such as those generated by *eld* operator, shown in Figure 46, give rise to unsat warnings during the constraint analysis procedure in the tool chain. Given that one or more clauses are just false, there are no solutions that could be found to allow the process to continue with a valid constraint. The usefulness of such operators might be subject to further inquiring.

Figure 46 – *Fibonacci* eld mutant extract

```
(...)
process Fibonacci ^=
begin
  state FibState == [x,y]
  InitFibState == [ x, y | false ]
  (...)
end
```

Source: Elaborated by the author.

Another sample that might be of interest is the output of the analysis for the *pcs* operator, which swaps two consecutive communications. Considering the communications inside *InitFib* action, the swap causes no semantic effect. The CFG analysis of such mutant yields the following output from the tool:

Analyzing 1 mutants for OPRpcs

OPRpcs#0: No diff found between CFGs. Maybe an equivalent mutant.

The procedure has not found any difference between the graphs, detecting a potential equivalent mutant. The warning only points out the situation as a possibility, given the unfeasibility of determining the equivalence of two specifications systematically.

5.2.4 Chronometer

The chronometer specification, shown in [Figure 3](#), was introduced by [Oliveira \(2006\)](#) and presented in [section 2.3](#). The machine readable version is also shown in [Figure 57](#). The very same specification was used for another *Circus* mutation testing experiment that was conceived with a different approach, as described in [section 5.3](#). It consisted in adapting and translating the *Chrono* specification to CSP, then apply a refinement checker to find traces counter examples that could be used to differentiate the original and mutated versions. These results were published in ([ALBERTO et al., 2016](#)), when the tool chain was not yet available and most of the theory validation was performed with almost no automation.

The set of mutants is limited in the CSP translation experiment, since there are syntactic constructions that are not allowed in CSP_M , e.g., the two mutants generated by event drop operator, as they would leave the external choice between the prefixed synchronizations of *tick* and *time* without a prefix. The operational semantics of *Circus* and our symbolic evaluator allow these constructs, postponing the choice of the continuation branch to future events or guards. As a consequence, the set of mutants in this tool chain validation experiment is wider than the one presented in ([ALBERTO et al., 2016](#)).

Moreover, the idea is not to compare the tool chain output with the CSP refinement checking experiments. It is important to state that the tool chain validation experiments aim for weak mutation, while the refinement checker aims, by definition, for strong mutation. Therefore, both approaches will yield necessarily different results. In [Table 6](#), we have the numbers of mutants generated for *Chrono* and their analysis in the tool chain.

5.3 Theory validation by CSP refinement checking

A way to validate the mutation testing theory for *Circus* is by translating, when possible ([OLIVEIRA; SAMPAIO; FILHO, 2014](#)), *Circus* specifications into CSP and making use of the well established model checking tools. Similar approaches for generating test cases using model checkers are popular; see, for instance, ([FRASER; WOTAWA; AMMANN, 2009](#)) for a survey. In our case, the checker is convenient to establish whether a generated mutant is a refinement of the original specification and, if not, to yield some counterexamples that provide a basis for building killer tests, as defined in [section 3.2](#).

Table 6 – Analysis of tool chain generated mutants for *Chrono*

Abbreviation	Operator	Mutants	Killed	Total dead configs.
ped	Event Drop	2	2	2
per	Event Replacement	2	2	2
pei	Event Insert	2	2	2
pco	Choice Operator	2	1	6
ppoSeqPar	Sequential Composition	2	2	2
ppoSeqInt	Interleave	2	2	2
pmr	Message Replacement	2	2	2
pcr	Channel Replacement	4	4	4
pci	Communication Insert	2	2	2
pce	Communication Elimination	2	2	2
pcs	Communication Swap	1	1	1
pprAction	Action Name	12	10	10
pprSchema	Schema Name	5	4	4
pprHide	Hide Events	5	3	3
eni	Negation Insert	7	4	4
eniGuard	Guard Negation	2	1	1
elr	Operator Replacement	3	2	2
eld	Operand Replacement	14	4	4
ear	Operator Replacement	6	3	3
eur	Unary Insertion	3	2	2
eak	Add k to Operand	3	2	2
esk	Sub k from Operand	3	2	2
err	Operator Replacement	40	25	25

Source: Research data.

The refinement checker FDR (FORMAL SYSTEMS (EUROPE) LTD, 1999) is a tool for analyzing specifications written in a particular machine readable CSP notation, namely CSP_M (SCATTERGOOD; ARMSTRONG, 2011). The tool has been used in production for over two decades and its development is active¹, given the frequent release of new versions, corrections and documentation updates. The refinement checker can be considered a mature tool for handling CSP models.

For *Circus*, there is no mature refinement model checker available at this time and is out of the scope of this work to develop one. Moreover, there are ongoing efforts (FREITAS, 2006; MOTA *et al.*, 2014) in this direction. However, for some *Circus* specifications, we can detour from this barrier via a translation of *Circus* models to CSP and benefit from the well established refinement checker for CSP_M .

This approach is applicable only to *Circus* specifications with simple data models that can be directly encoded in CSP without further refinement. It has been, in any case, very useful to validate our ideas and approach, specially before the construction of our symbolic evaluator. We discuss below the translation from *Circus* to CSP and the test-generation technique.

¹ <https://www.cs.ox.ac.uk/projects/fdr/>

Table 7 – Analysis of *Chrono* mutants in FDR

Abbreviation	Operator	Mutants	Killed by FDR
ped	Event Drop	1	1
per	Event Replacement	2	2
pei	Event Insert	2	2
pco	Choice Operator	2	2
ppoSeqPar	Sequential Composition	1	1
ppoSeqInt	Interleave	1	1
pmr	Message Replacement	2	2
pci	Communication Insert	1	1
pce	Communication Elimination	1	1
pprSchema	Schema Name	6	6
eniGuard	Guard Negation	2	2
eld	Operand Replacement	4	4
ear	Operator Replacement	1	1
eur	Unary Insertion	2	2
eak	Add k to Operand	6	6
esk	Sub k from Operand	6	6
ead	Operand Replacement	6	6
err	Operator Replacement	10	8

Source: Research data.

5.3.1 Translating *Circus* specifications into CSP

As shown in (OLIVEIRA; SAMPAIO; FILHO, 2014), a subset of *Circus* that does not use complex Z specifications can be automatically translated to CSP. Conveniently, it is the case for the *Chrono* example and the mutants we derived from it. To work with FDR, it is necessary to write specifications using the machine readable dialect of CSP (SCATTERGOOD; ARMSTRONG, 2011) following the guidelines below:

- Channels and types are kept;
- The state components are encapsulated into a single data type with constructor *AState*, and passed as parameter to all processes and functions;
- A *Circus* action is translated into a CSP process;
- A reference to a schema s in an action A is unfolded to an intermediary process $A_s(s(AState))$;
- Loops are translated as tail recursive calls;
- State changes defined by Z operations are performed by CSP functions from *AState* to *AState*.

For instance, Figure 47 shows the translation of *Chrono*. The *AInit*, *IncSec* and *IncMin* *Circus* schemas are translated to the *ainit*, *incsec* and *incmin* functions. Arguments and

results of these functions are yielded by the type constructor *AState*, encapsulating values for the two state components *sec* and *min*. The anonymous main action of *Chrono* is translated into the *CHRONO* CSP process. The initialisation is captured by the parameter of the call to process *RUN*. The loop is captured by parameterised recursive calls at the end of each choice of *RUN*. The references to the schemas *IncSec* and *IncMin* inside the *Run* action are unfolded into the sub-process *RUN_incsec* and *RUN_incsec_incmin*.

5.3.2 Using FDR for fault-based test-case generation

In subsection 3.2.1 we have explained that counterexamples yielded by a refinement check of a mutant against the original specification provide bases to build tests to kill the mutant. Figure 48 shows the output of FDR for the check $CHRONO \sqsubseteq_F MCHRONO$ for failures refinement in the context of the definitions in Figure 47. It defines the following trace *s*, forbidden continuation *a*, and acceptance set *X*.

$$s = \langle tick, time \rangle \quad a = out.1.1 \quad X = \{out.0.1\}$$

For this experiment, the production of a counterexample gives rise to our account of the mutant as killed in Table 7. As stated in Theorem 1, finding such counterexamples is enough to assure that *MCHRONO* is a mutant of interest: it gives us the necessary elements to build a test case for, at least, the failures fault-based set.

In the particular example of the $CHRONO \sqsubseteq_F MCHRONO$ analysis, we have both a failure and a forbidden continuation, allowing the construction of one test for each of the $FBTests_T^{CHRONO}(MCHRONO)$ and $FBTests_F^{CHRONO}(MCHRONO)$ sets. Such tests are constructed using the T_T and T_F functions.

$$\begin{aligned} T_T(s, a) &= inc \rightarrow tick \rightarrow inc \rightarrow time \rightarrow pass \rightarrow out.1.1 \rightarrow fail \rightarrow \text{Stop} \\ T_F(s, X) &= inc \rightarrow tick \rightarrow fail \rightarrow time \rightarrow fail \rightarrow out.0.1 \rightarrow pass \rightarrow \text{Stop} \end{aligned}$$

In the traces of the execution of a parallel composition between the mutant *MCHRONO* and any of these tests the last verdict event is *fail*, thus the tests kill the mutant.

All mutants generated for *Chrono* (in Figure 3) have been translated to CSP and analysed with FDR. As shown in Table 7, only two of the mutants are not of interest. Both are generated by the relational operator replacement *err*, and are in Figure 49. The changes introduced by *err* in both *ChronoErrEquiv1* and *ChronoErrEquiv2* occur in the expressions guarding the external choice between *IncMin* and *Skip*. In *ChronoErrEquiv1*, the original guard ($sec = 0$) is replaced with ($sec \leq 0$), and for *ChronoErrEquiv2*, the guard ($sec \neq 0$) is replaced with ($sec > 0$). Both changes cause no observable effect, since *Range* imposed lower boundary to *sec* is 0.

Figure 47 – Chrono in CSP

```

---- The chronometer in CSP_M
--Type and channels declaration
Range = 0..59
channel tick, time
channel out:Range.Range

--State simulation with single data type
Minsec = (min,sec) | min <- Range, sec <- Range
datatype Clock = AState.Minsec

--AInit schema translation: state initialization min=0 and sec=0
ainit() = AState.(0,0)
--IncSec schema translation: increments sec by one within Range
incsec(AState.(min,sec)) = AState.(min,(sec+1)%60)
--IncMin schema translation: increments min by one within Range
incmin(AState.(min,sec)) = AState.((min+1)%60,sec)

--Anonymous action to RUN process, with ainit state inialization
CHRONO = RUN(ainit())
--Run action translation, stateful loop achieved using recursion
RUN(AState.(min,sec)) =
  tick -> RUN_incsec(incsec(AState.(min,sec)))
  []
  time -> out.min.sec -> RUN(AState.(min,sec))
--Run sub-process translation, applying incsec function
RUN_incsec(AState.(min,sec)) =
  (sec == 0) & RUN_incsec_incmin(incmin(AState.(min,sec)))
  []
  (sec != 0) & RUN(AState.(min,sec))
RUN_incsec_incmin(AState.(min,sec)) = RUN(AState.(min,sec));

--First mutant generated with eniGuard operator
MCHRONO = MRUN(ainit())
MRUN(AState.(min,sec)) =
  tick -> MRUN_incsec(incsec(AState.(min,sec)))
  []
  time -> out.min.sec -> MRUN(AState.(min,sec))
MRUN_incsec(AState.(min,sec)) =
  --Negated guard mutation introduced below
  (not sec == 0) & MRUN_incsec_incmin(incmin(AState.(min,sec)))
  []
  (sec != 0) & MRUN(AState.(min,sec))
MRUN_incsec_incmin(AState.(min,sec)) = MRUN(AState.(min,sec));

```

Source: Elaborated by the author.

Figure 48 – FDR output for $CHRONO \sqsubseteq_F MCHRONO$

```

Result: Failed
Visited States: 7
Visited Transitions: 17
Visited Plys: 2
Estimated Total Storage: 67MB
Counterexample (Trace Counterexample)
  Specification Debug:
    Trace: <tick, time>
    Available Events: out.0.1
  Implementation Debug:
    MRUN(AState.(0, 0)) (Trace Behaviour):
      Trace: <tick, time>
      Error Event: out.1.1

```

Source: Elaborated by the author.

Figure 49 – Two equivalent mutants of *Chrono* exposed by FDR

```

process ChronoErrEquiv1  $\hat{=}$ 
...
  Run  $\hat{=}$  (tick  $\rightarrow$  IncSec; ((sec  $\leq$  0) & IncMin)
                                $\square$ ((sec  $\neq$  0) & Skip)))
         $\square$ (time  $\rightarrow$  out!(min, sec)  $\rightarrow$  Skip)
        • (AInit; ( $\mu$  X • (Run; X)))

process ChronoErrEquiv2  $\hat{=}$ 
...
  Run  $\hat{=}$  (tick  $\rightarrow$  IncSec; ((sec = 0) & IncMin)
                                $\square$ ((sec > 0) & Skip)))
         $\square$ (time  $\rightarrow$  out!(min, sec)  $\rightarrow$  Skip)
        • (AInit; ( $\mu$  X • (Run; X)))

```

Source: Elaborated by the author.

5.4 Validating the symbolic execution tool

This section presents some examples of *Circus* specifications that we used for checking the resulting traces from our K-Framework implementation against the ones expected following the formal operational semantics.

The first example, shown in Figure 50, specifies a *Circus* process containing in the main anonymous action a parallel composition with a defined synchronization set and two concurrent external choices. The initials of events from channels that are not elements of the synchronization

set are forwarded as soon as they are posted. The non-deterministic search finds all the possible solutions, including the partial ones, i.e., traces which do not reach the final Skip.

Figure 50 – Synchronization test specification

```
channel a1,a2,a3,b1,b2,b3,c1,c2
channelset SyncSet == {|a1,b1|}
process proc3 ^=
begin
  @
  ((b2 -> b3 -> Skip) [] (a1 -> a2 -> a3 -> Skip))
  [|SyncSet|]
  ((c1 -> b1 -> Skip) [] (c2 -> a1 -> Skip))
end
```

Source: Elaborated by the author.

The output is shown in [Figure 51](#), listing all the solutions found and combining all the possible event orders caused by the non-determinism. The Solution 6 shows the only trace that reaches the final Skip.

Figure 51 – Synchronization test output result

```
Search results:
Const:Bool --> true

Solution 1: b2 ~> b3 ~> c1
Solution 2: b2 ~> b3 ~> c2
Solution 3: b2 ~> c1 ~> b3
Solution 4: b2 ~> c2 ~> b3
Solution 5: c1 ~> b2 ~> b3
Solution 6: c2 ~> a1 ~> a2 ~> a3 ~> Skip
Solution 7: c2 ~> b2 ~> b3
```

Source: Elaborated by the author.

The specification shown in [Figure 52](#) exercises the variable scope handling. The scope of the input variable *x* is checked before and after the redeclaration that occurs inside the named action *Act*.

Figure 52 – Variable scope test specification

```

channel c
process Scope ^=
begin
  Act ^= c?x -> c!x -> Skip
  @
  c?x -> (c!x -> Act; c!x -> Skip)
end

```

Source: Elaborated by the author.

As defined in the operational semantics, the symbolic value assigned to a previously declared variable name is restored with the ending of the scope that contains the redeclaration. The output is shown in [Figure 53](#), including the constraint and tracking of scope ending in the specification trace.

Figure 53 – Scope test output result

```

Search results:
Solution 1:
Const:Bool -->
V0 ==K V1 andBool V2 ==K V3
SpecTrace:K -->
( c ?? x ) ~> ( c !! x ) ~> ( c ?? x ) ~> ( c !! x )
~> ( end x ) ~> ( c !! x ) ~> end x
Trace:K -->
( c ?? V1 ) ~> ( c ! V0 ) ~> ( c ?? V3 ) ~> ( c ! V2 )
~> ( c ! V0 )

```

Source: Elaborated by the author.

In [Figure 54](#) we show a specification that exercises the concurrent assignment to state variables. The main action begins by assigning 0 to all the three variable names. Then it splits into two parallel actions: the left part is allowed to write to x and the right part is allowed to write to z. The left and right parts assigns, respectively, 1 and 2 to all the three variable names. The parallel composition finishes with the assignments and the main action continues communicating the values of the three variables through the channel out.

Figure 54 – Parallel state assignment specification text

```

channel out
process ParState ^=
begin
  state myState == [x,y,z]
  @
  x:=0;y:=0;z:=0;
  ((x:=1;y:=1;z:=1) [|{|x|}|{|}|{|y|}|] (x:=2;y:=2;z:=2));
  out!x -> out!y -> out!z -> Skip
end

```

Source: Elaborated by the author.

The generated output can be seen in [Figure 55](#), showing a result accordingly to the expected: the sequence of the three out communications containing the values of x, y and z, respectively, 1, 2 and 0. It means that x and y were changed by the left and right parts of the parallel composition while z remained with its original assignment. The output is given by symbolic values that are referenced in the constraint Const.

Figure 55 – Parallel state assignment output result

```

Search results:
Solution 1:
Const:Bool -->
V1 ==K 0 andBool V0 ==K 1 andBool V2 ==K 2
SpecTrace:K -->
( x := 0 ) ~> ( y := 0 ) ~> ( z := 0 ) ~> ( x := 1 )
~> ( x := 2 ) ~> ( y := 2 ) ~> ( z := 2 ) ~> ( y := 1 )
~> ( z := 1 ) ~> ( out !! x ) ~> ( out !! y ) ~> ( out !! z )
Trace:K -->
( out ! V0 ) ~> ( out ! V2 ) ~> ( out ! V1 ) ~> Skip

```

Source: Elaborated by the author.

The example *Circus* process already shown in [Figure 45](#) specifies the behavior of a Fibonacci sequence generator as introduced in ([CAVALCANTI; MOTA; WOODCOCK, 2013](#)). Since OutFib is a recursive action, the traces are unbounded and it is necessary to limit the number of rewrite steps we are going to apply in the evaluation. In [Figure 56](#) we present the output arbitrarily limited to 500 rewrite steps. The resulting trace is consistent with the operational semantics, as it was discussed in ([CAVALCANTI; MOTA; WOODCOCK, 2013](#)).

Figure 56 – Fibonacci output result

```

Search results:
Solution 1:
Const:Bool -->
true
SpecTrace:K -->
( out ! 1 ) ~> ( out ! 1 ) ~> InitFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState ~> (out !! ( x + y )) ~> OutFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState ~> (out !! ( x + y )) ~> OutFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState ~> (out !! ( x + y ))
Trace:K -->
( out ! 1 ) ~> ( out ! 1 ) ~> ( out ! 2 ) ~> ( out ! 3 )
~> ( out ! 5 ) ~> ( out ! 8 ) ~> ( out ! 13 ) ~> ( out ! 21 )
~> ( out ! 34 ) ~> ( out ! 55 ) ~> ( out ! 89 )

```

Source: Elaborated by the author.

The chronometer specification, shown in [Figure 57](#), was introduced in [section 2.3](#). Here, we present a version including secondary processes to compose a working clock. It contains three *Circus* processes: the *Chrono*, that keeps in its state the count of elapsed minutes and seconds; the *Ticker*, that synchronizes with *Chrono* in the *tick* channel, causing the increment of seconds, and in *time* channel, causing the output of the current number of elapsed minutes and seconds; and *Clock*, that parallel composes *Chrono* and *Ticker*, setting the synchronization channels and hiding the output of the channel *tick*.

Figure 57 – *Chrono* specification text

```

channel out,tick,time

process Chrono ^=
begin
  state AState == [sec,min]
  AInit == [min,sec | sec' == min' && min' == 0]
  IncSec == [min,sec | sec' == ((sec+1) mod 60) && min' == min]
  IncMin == [min,sec | min' == ((min+1) mod 60) && sec' == sec]
  Run ^= (tick -> IncSec;
          (((sec==0) & IncMin) [] ((sec != 0) & Skip))
        )
        []
        (time -> out!min -> out!sec -> Skip)
  @
  AInit; u X @ (Run; X)
end

process Ticker ^= u X @ (tick -> time; X)
process Clock ^= (Chrono [|{|tick,time|}|] Ticker) /H {|tick|}

:run Clock

```

Source: Elaborated by the author.

The line `:run Clock` defines *Clock* as the process to be evaluated. We shown the resulting output in [Figure 58](#), that is correct in consideration to the operational semantics analysis presented in ([ALBERTO et al., 2016](#)).

Figure 58 – Chrono output result

```

Search results:
Solution 1:
Const:Bool -->
true
SpecTrace:K -->
AInit ~> IncSec ~> ( sec != 0 ) ~> time ~> ( out !! min )
~> ( out !! sec ) ~> IncSec ~> ( sec != 0 ) ~> time
~> ( out !! min ) ~> ( out !! sec ) ~> IncSec ~>
(...)
~> time ~> ( out !! min ) ~> ( out !! sec ) ~> IncSec
~> ( sec == 0 ) ~> IncMin ~> time ~> ( out !! min )
~> ( out !! sec ) ~> IncSec ~> ( sec != 0 ) ~> time
~> ( out !! min ) ~> ( out !! sec ) ~> (...)
Trace:K -->
time ~> ( out ! 0 ) ~> ( out ! 1 ) ~> time
~> ( out ! 0 ) ~> ( out ! 2 ) ~> time ~> ( out ! 0 )
~> ( out ! 3 ) ~> time ~> ( out ! 0 ) ~>
(...)
~> time ~> ( out ! 0 ) ~> ( out ! 59 ) ~> time
~> ( out ! 1 ) ~> ( out ! 0 ) ~> time ~> ( out ! 1 )
~> ( out ! 1 ) ~> (...)

```

Source: Elaborated by the author.

5.5 Final considerations

We have presented the experiments that were used to validate the automated tools for mutation testing in *Circus*. Despite the fact that all implementations are in a premature release stage, the results have shown that it is possible to use a certain level of automation to construct fault-based tests cases for *Circus* specifications. More development effort is required to achieve better interfaces and more transparent communication between the modules. Industry scale tests would be desirable and will be included in future works.

CONCLUSIONS AND FUTURE WORK

In the realm of software engineering, specification and test of systems benefits largely from the precision and rigor of formal models. Modeling techniques, such as process algebras, are utterly useful to describe and analyze the behaviour of systems and processes. This work has been directed for investigating fault based test to the process algebra *Circus*: a state-rich modeling language combining Z (WOODCOCK; DAVIES, 1996), CSP (ROSCOE, 1998), and a refinement calculus (MORGAN, 1994).

Coverage-based testing was available for *Circus* since (CAVALCANTI; GAUDEL, 2014), paving the road for investigations on how fault-based techniques might be used for the benefit of practical testing procedures. We have proposed the use of mutation testing techniques to establish criteria for generating test cases that target specific faults in *Circus* models.

6.1 Contributions

We have formally defined mutation testing for *Circus* by characterizing (1) the exhaustive sets of tests that can kill a given mutant, considering both traces refinement and *conf*, that is, process refinement in *Circus* as a whole; (2) a technique to select tests from these sets based on specification traces of the mutants; and (3) an extensive collection of operators to generate mutants considering faults related to both reactive and data manipulation behaviour. Like the *Circus* testing theory, this work is of general relevance for state-rich algebras for refinement (ALBERTO *et al.*, 2016). Even though mutation testing has already been applied to languages and theories upon which *Circus* is founded, the consideration of a state-rich process algebra for refinement with a UTP semantics is novel.

In the course of building the necessary tools to perform automated mutation test for *Circus*, we have stumbled upon a bigger challenge: a symbolic evaluator for the language was not available. We have used a rewriting logic framework to automate the behavior of the transition

system defined by the formal operational semantics of *Circus*. We also embedded a subset of the operational semantics for *Circus* specification traces (ALBERTO; GAUDEL, 2017). The result is an executable interpreter, which accepts a specification text as input and enumerates its constrained symbolic traces and specification traces. Presently, the tool is the first symbolic specification traces generator for the *Circus* language.

A tool chain to aid in the automation of mutation test for *Circus* was also introduced as a prototype, including modules to manipulate the language abstract syntax tree, build control flow graphs, inspect the mutation and slice specification texts. Heuristic algorithms were designed to perform the analysis and provide paths that are likely to reach the seeded fault, allowing the generation of traces that can be used in building test cases to kill mutants.

6.2 Limitations

Considering the symbolic evaluator, most of the rewrite rules are very close to their counterparts in the operational semantics. Nevertheless, some adaptations were needed and they complicate the establishment of the correctness of the implementation. It would be interesting to conduct some proofs over the adopted strategies.

The current version of the symbolic trace generator is limited in dealing with certain aspects of *Circus* specifications: type checking, multiple valuated communication and schema expressions. A full account of the limitations and the ideas to overcome them is reported in (ALBERTO; GAUDEL, 2017).

The modules of the tool chain are not ready for production, given that the user and inter-module interfaces need further refinement. Also, the interface limitations increased the time necessary for elaborating experiments, restricting the validation to a few samples. More work is needed to achieve a user friendly level and allow the experimentation with a larger number of specifications.

6.3 Future work

Given a mutation operator such as one of those from section 3.3, it is attractive to avoid constructing the mutants to generate the tests. For that, we need a way of calculating the traces and failures of the mutants without constructing them. This calculation could take advantage of the knowledge of traces and failures of the original process, but the effect of the mutation operators on the semantics of a process is not direct. More precisely, the semantic models of interest are not a congruence for the mutant operators. This gives rise to an interesting challenge.

Generally, mutations are syntactic changes. Since the specification traces are close to the syntax of the specifications, it may well be the case that they do provide an adequate way to construct traces of the mutants in terms of those of the original specification. An analysis of

mutation operators for specification traces and their relation to our operators is an interesting avenue for future work as well.

More experiments are desirable to validate the automation tools. We plan to improve the stage of development of the software that was produced, specially the user interface, facilitating the design and execution of new experiments. Once a level adequate for production environments is achieved, the new results should be presented in a tool oriented publication.

As said above, given that adaptations over the formal semantics were necessary to implement the behaviour of concurrent process synchronizations in rewriting rules, the symbolic evaluator should be subjected to proofs of correctness.

The rule set developed for dealing with *Circus* included significant parts of CSP and $\mathbb{Z}$ semantics, and it would be an interesting exercise to reuse the relevant parts of this set to implement the complete operational semantics for both specification languages.

Beyond providing interpreters based on formal semantics, the symbolic evaluator framework has proven to be adequate for other static analysis and state-exploration purposes: once introduced the semantics, it is possible to extend the configuration and the structural rules in order to develop a wide class of tools. Moreover, we believe that this piece of work brings some insights to the automation of process algebra operational semantics in general.

Acknowledgments

We are grateful to the Brazilian Funding Agency CNPq (Grant 400834/2014-6) for supporting this project, and to LRI (Laboratoire de Recherche en Informatique, Univ. Paris-Sud and CNRS) for hosting Alex Alberto during part of this work.

BIBLIOGRAPHY

ABRIAL, J.-R. The specification language z: Syntax and semantics. **Programming Research Group, Oxford Univ**, 1980. Citation on page 29.

_____. **The B-Book: assigning programs to meanings**. [S.l.: s.n.], 1996. Citations on pages 27 and 29.

ADRION, W. R.; BRANSTAD, M. A.; CHERNIAVSKY, J. C. Validation, verification, and testing of computer software. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 14, n. 2, p. 159–192, 1982. ISSN 0360-0300. Citation on page 21.

AGRAWAL, H.; DEMILLO, R.; HATHAWAY, R.; HSU, W.; HSU, W.; KRAUSER, E.; MARTIN, R. J.; MATHUR, A.; SPAFFORD, E. **Design of mutant operators for the C programming language**. [S.l.], 1989. Citation on page 55.

AICHERNIG, B.; JIFENG, H. Mutation testing in UTP. **Formal Aspects of Computing**, v. 21, p. 33–64, 2008. Citation on page 61.

ALBERTO, A.; CAVALCANTI, A.; GAUDEL, M.-C.; SIMÃO, A. Formal mutation testing for circus. **Information and Software Technology**, Elsevier, 2016. Citations on pages 24, 61, 97, 121, 131, and 133.

ALBERTO, A.; GAUDEL, M.-C. **Defining the Circus operational semantics in the K-framework**. [S.l.], 2017. Citations on pages 24, 110, and 134.

ALJAZZAR, H.; LEUE, S. K*: A heuristic search algorithm for finding the k shortest paths. **Artificial Intelligence**, Elsevier, v. 175, n. 18, p. 2129–2154, 2011. Citation on page 93.

ALLEN, F. E. Control flow analysis. In: ACM. **ACM Sigplan Notices**. [S.l.], 1970. v. 5, n. 7, p. 1–19. Citation on page 86.

ALLEN, R. J. **A Formal Approach to Software Architecture**. [S.l.], 1997. Citation on page 27.

ANDREWS, Z.; FITZGERALD, J.; PAYNE, R.; ROMANOVSKY, A. Fault modelling for systems of systems. In: **Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on**. [S.l.: s.n.], 2013. p. 1–8. Citation on page 78.

ANDREWS, Z.; PAYNE, R.; ROMANOVSKY, A.; DIDIER, A.; MOTA, A. Model-based development of fault tolerant systems of systems. In: **Systems Conference (SysCon), 2013 IEEE International**. [S.l.: s.n.], 2013. p. 356–363. Citation on page 78.

BAETEN, J. C. M. A brief history of process algebra. **Theor. Comput. Sci.**, Elsevier Science Publishers Ltd., Essex, UK, v. 335, n. 2-3, p. 131–146, May 2005. ISSN 0304-3975. Available: <<http://dx.doi.org/10.1016/j.tcs.2004.07.036>>. Citation on page 30.

BALDWIN, D.; SAYWARD, F. **Heuristics for Determining Equivalence of Program Mutations**. [S.l.], 1979. Citation on page 58.

BARROCA, L. M.; MCDERMID, J. A. Formal methods: Use and relevance for the development of safety-critical systems. **The Computer Journal**, v. 35, n. 6, p. 579–599, 1992. Citation on page 26.

BARROCAS, S. L. M.; OLIVEIRA, M. V. M. A Validation Strategy for an Automatic Code Generator using Java Pathfinder. In: ARABNIA, H. R.; DELIGIANNIDIS, L.; TINETTI, F. G. (Ed.). **Proceedings of the 14th International Conference on Software Engineering Research and Practice - SERP 2016**. [S.l.]: CSREA Press, 2016. p. 277–283. ISBN 1-60132-446-4. Citation on page 27.

BASILI, V. R.; PERRICONE, B. T. Software errors and complexity: an empirical investigation. **Communications of the ACM**, ACM, v. 27, n. 1, p. 42–52, 1984. Citation on page 56.

BERGSTRA, J. A. **Handbook of Process Algebra**. New York, NY, USA: Elsevier Science Inc., 2001. ISBN 0444828303. Citation on page 30.

BERNOT, G.; GAUDEL, M. C.; MARRE, B. Software testing based on formal specifications: a theory and a tool. **Software Engineering Journal**, IET, v. 6, n. 6, p. 387–405, 1991. Citation on page 46.

BLACK, P. E.; OKUN, V.; YESHA, Y. Mutation operators for specifications. In: IEEE. **Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on**. [S.l.], 2000. p. 81–88. Citations on pages 23, 54, 69, and 77.

BOCHMANN, G. V.; PETRENKO, A. Protocol testing: review of methods and relevance for software testing. In: **ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis**. New York, NY, USA: ACM, 1994. p. 109–124. ISBN 0-89791-683-2. Citation on page 21.

BOG, A.; PUHLMANN, F.; WESKE, M. The piviztool: Simulating choreographies with dynamic binding. In: ADAMS, M.; SADIQ, S. W. (Ed.). **BPM (Demos)**. CEUR-WS.org, 2007. (CEUR Workshop Proceedings, v. 272). Available: <<http://dblp.uni-trier.de/db/conf/bpm/bpmd2007.htmlBogPW07>>. Citation on page 33.

BOWEN, J. The ethics of safety-critical systems. **Communications of the ACM**, ACM, v. 43, n. 4, p. 91–97, 2000. Citation on page 26.

BOWEN, J.; STAVRIDOU, V. The industrial take-up of formal methods in safety-critical and other areas: A perspective. In: SPRINGER. **FME'93: Industrial-Strength Formal Methods**. [S.l.], 1993. p. 183–195. Citation on page 26.

_____. Safety-critical systems, formal methods and standards. **Software Engineering Journal**, IET, v. 8, n. 4, p. 189–209, 1993. Citation on page 26.

BOWEN, J. P. **Formal specification and documentation using Z: A case study approach**. [S.l.]: International Thomson Computer Press London, 1996. Citation on page 37.

BOWEN, J. P.; HINCHEY, M. G. Seven more myths of formal methods. **Software, IEEE**, IEEE, v. 12, n. 4, p. 34–41, 1995. Citation on page 26.

_____. Ten commandments of formal methods. **Computer**, IEEE, v. 28, n. 4, p. 56–63, 1995. Citation on page 26.

BRINKSMA, E. A theory for the derivation of tests. In: **Protocol Specification, testing and Verification VIII**. [S.l.]: North-Holland, 1988. p. 63–74. Citation on page 51.

BUDD, T. A. **Mutation Analysis of Program Test Data**. Phd Thesis (PhD Thesis) — Yale University, New Haven, CT, USA, 1980. AAI8025191. Citation on page 83.

BUDD, T. A.; ANGLUIN, D. Two notions of correctness and their relation to testing. **Acta Inf.**, v. 18, p. 31–45, 1982. Citations on pages 44, 45, 57, 58, and 86.

CAVALCANTI, A. L. C.; CLAYTON, P.; O'HALLORAN, C. From Control Law Diagrams to Ada via *Circus*. **Formal Aspects of Computing**, v. 23, n. 4, p. 465–512, 2011. Citations on pages 22 and 38.

CAVALCANTI, A. L. C.; GAUDEL, M.-C. Testing for refinement in csp. In: **Formal Methods and Software Engineering**. [S.l.]: Springer, 2007. p. 151–170. Citations on pages 46, 48, 49, 50, 51, 52, and 117.

_____. A note on traces refinement and the *conf* relation in the Unifying Theories of Programming. In: **Unifying Theories of Programming 2008**. [S.l.]: Springer, 2010. (LNCS, v. 5713), p. 42–61. Citation on page 51.

_____. Specification Coverage for Testing in *Circus*. In: **Unifying Theories of Programming**. [S.l.]: Springer, 2010. (LNCS, v. 6445), p. 1–45. Citations on pages 65 and 182.

_____. Testing for Refinement in *Circus*. **Acta Informatica**, v. 48, n. 2, p. 97–147, 2011. Citations on pages 23, 51, 52, 53, 54, 64, 65, 96, 100, and 105.

_____. **Data Flow Coverage of Circus Specifications-Extended Version**. Université Paris-Sud XI, 2013. Citations on pages 110 and 158.

_____. Data Flow coverage for *Circus*-based testing. In: **FASE 2014**. [S.l.: s.n.], 2014. (LNCS, v. 8441), p. 415–429. Citations on pages 23, 54, 65, 66, 68, 96, and 133.

CAVALCANTI, A. L. C.; MOTA, A.; WOODCOCK, J. C. P. Simulink timed models for program verification. In: LIU, Z.; WOODCOCK, J. C. P.; ZHU, H. (Ed.). **Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday**. [S.l.]: Springer, 2013. (Lecture Notes in Computer Science, v. 8051), p. 82–99. Citations on pages 118 and 129.

CHOW, T. S. Testing Software Design Modeled by Finite-State Machines. **IEEE Trans. Software Eng.**, v. 4, n. 3, p. 178–187, May 1978. Citations on pages 27 and 29.

CIOBACA, S.; LUCANU, D.; RUSU, V.; ROSU, G. A language-independent proof system for full program equivalence. **Formal Aspects of Computing**, v. 28, n. 3, p. 469–497, May 2016. Citation on page 96.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. **ACM Computing Surveys (CSUR)**, ACM, v. 28, n. 4, p. 626–643, 1996. Citations on pages 26 and 28.

CRAIGEN, D.; GERHART, S.; RALSTON, T. Formal methods reality check: Industrial usage. **Software Engineering, IEEE Transactions on**, IEEE, v. 21, n. 2, p. 90–98, 1995. Citation on page 26.

DAVIES, J. Using csp. In: **Refinement Techniques in Software Engineering**. [S.l.]: Springer, 2006. p. 64–122. Citation on page [37](#).

DELAMARO, M. E.; MALDONADO, J. C. Proteum-A tool for the assessment of test adequacy for C programs. In: **Proceedings of the Conference on Performability in Computing Systems (PCS 96)**. New Brunswick, NJ: [s.n.], 1996. p. 79–95. Citation on page [84](#).

DELAMARO, M. E.; PEZZÈ, M.; VINCENZI, A. M. R.; MALDONADO, J. C. Mutant operators for testing concurrent Java programs. In: **SBES'01 - Simpósio Brasileiro de Engenharia de Software**. Rio de Janeiro/RJ - Brasil: [s.n.], 2001. p. 272–285. Citation on page [84](#).

DEMILLO, R. A.; GUINDI, D. S.; MCCRACKEN, W.; OFFUTT, A. J.; KING, K. An extended overview of the mothra software testing environment. In: IEEE. **Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on**. [S.l.], 1988. p. 142–151. Citation on page [84](#).

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: help for the practicing programmer. **IEEE Computer**, v. 11, n. 4, p. 34–41, April 1978. Citations on pages [54](#) and [55](#).

DEMILLO, R. A.; OFFUTT, A. J. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, v. 17, n. 9, p. 900–910, Sep 1991. ISSN 0098-5589. Citations on pages [58](#) and [86](#).

DIJKSTRA, E. W. **Structured programming**. London, UK, UK: Academic Press Ltd., 1972. 1–82 p. ISBN 0-12-200550-3. Citation on page [42](#).

_____. Guarded commands, nondeterminacy and formal derivation of programs. **Communications of the ACM**, ACM, v. 18, n. 8, p. 453–457, 1975. Citations on pages [28](#) and [34](#).

FELIACHI, A.; GAUDEL, M.-C.; WOLF, B. **Exhaustive testing in HOL-TestGen/CirTA - A case study**. [S.l.], 2013. Citation on page [98](#).

FLOYD, R. W. Assigning meanings to programs. **Mathematical aspects of computer science**, v. 19, n. 19-32, p. 1, 1967. Citations on pages [27](#) and [28](#).

FORMAL SYSTEMS (EUROPE) LTD. **FDR: User Manual and Tutorial, version 2.28**. [S.l.], 1999. Citation on page [122](#).

FRANKL, P. G.; WEYUKER, E. J. Testing software to detect and reduce risk. **Journal of Systems and Software**, v. 53, p. 275–286, Sep 2000. Citation on page [44](#).

FRASER, G.; WOTAWA, F.; AMMANN, P. E. Testing with model checkers: a survey. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 19, n. 3, p. 215–261, 2009. Citation on page [121](#).

FREITAS, A. **From Circus to Java: Implementation and Verification of a Translation Strategy**. Master's Thesis (Master's Thesis) — Department of Computer Science, The University of York, dec 2005. Citation on page [27](#).

FREITAS, L.; MCDERMOTT, J. P. Formal methods for security in the xenon hypervisor. **Int. Jal on Software Tools for Technology Transfer**, v. 13, n. 5, p. 463 – 489, 2011. Citations on pages [22](#) and [39](#).

FREITAS, L. J. S. **Model Checking Circus**. Phd Thesis (PhD Thesis) — University of York, Department of Computer Science, 2006. Citation on page [122](#).

FUJIWARA, S.; BOCHMANN, G. v. Testing non-deterministic state machines with fault coverage. In: **Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV**. [S.l.]: North-Holland Publishing Co., 1991. p. 267–280. Citation on page [52](#).

FUJIWARA, S.; BOCHMANN, G. V.; KHENDEK, F.; MOKHTAR, A.; GHEDAMSI, A. Test selection based on finite state models. **IEEE Transactions on Software Engineering**, v. 17, n. 6, p. 13, Jun 1991. Citation on page [51](#).

GAUDEL, M.-C. Testing can be formal, too. In: **Int. Joint Conf., Theory And Practice of Software Development**. [S.l.]: Springer, 1995. (LNCS, v. 915), p. 82–96. Citations on pages [23](#), [46](#), [51](#), and [54](#).

GAUDEL, M.-C.; JAMES, P. J. Testing algebraic data types and processes : a unifying theory. **Formal Aspects of Computing**, v. 10, n. 5-6, p. 436–451, 1998. Citations on pages [23](#) and [51](#).

GEIST, R.; OFFUTT, A. J.; HARRIS, F. C. Estimation and enhancement of real-time software reliability through mutation analysis. **IEEE Transactions on Computers**, IEEE, v. 41, n. 5, p. 550–558, 1992. Citation on page [55](#).

GHEZZI, C.; KEMMERER, R. A. Astral: an assertion language for specifying realtime systems. In: **ESEC'91**. [S.l.]: Springer, 1991. p. 122–146. Citation on page [29](#).

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. W. FDR3 — A Modern Refinement Checker for CSP. In: **Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.: s.n.], 2014. p. 187–201. Citation on page [62](#).

GOODENOUGH, J. B.; GERHART, S. L. Toward a theory of test data selection. In: **Proceedings of the international conference on Reliable software**. New York, NY, USA: ACM, 1975. p. 493–510. Citation on page [43](#).

HALL, A. Seven myths of formal methods. **Software, IEEE**, IEEE, v. 7, n. 5, p. 11–19, 1990. Citation on page [26](#).

HAREL, D. Statecharts: A visual formalism for complex systems. **Science of Computer Programming**, v. 8, n. 3, p. 231–274, June 1987. Citations on pages [26](#) and [29](#).

HAREL, D.; PNUELI, A. On the development of reactive systems. Springer-Verlag New York, Inc., New York, NY, USA, p. 477–498, 1985. Citations on pages [22](#) and [26](#).

HARMAN, M.; HIERONS, R.; DANICIC, S. The relationship between program dependence and mutation analysis. In: **Mutation testing for the new century**. [S.l.]: Springer, 2001. p. 5–13. Citation on page [58](#).

HAYES, I.; FLINN, B. **Specification case studies**. [S.l.]: Prentice-Hall International London, 1987. Citation on page [37](#).

HIERONS, R. M.; JOURDAN, G.-V.; URAL, H.; YENIGÜN, H. Checking sequence construction using adaptive and preset distinguishing sequences. In: **Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009**. [S.l.: s.n.], 2009. p. 157–166. Citation on page [22](#).

HOARE, C. A. R. An axiomatic basis for computer programming. **Communications of the ACM**, ACM, v. 12, n. 10, p. 576–580, 1969. Citation on page 28.

_____. Communicating sequential processes. **Communications of the ACM**, ACM, v. 21, n. 8, p. 666–677, 1978. Citations on pages 27, 34, and 35.

_____. **Communicating sequential processes**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5. Citations on pages 34 and 107.

_____. **Proof of correctness of data representations**. [S.l.]: Springer, 2002. Citation on page 27.

HOARE, C. A. R.; JIFENG, H. **Unifying Theories of Programming**. [S.l.]: Prentice-Hall, 1998. Citations on pages 22, 38, and 40.

HOLZMANN, G. Design and validation of computer protocols prentice hall. **New Jersey**, 1991. Citation on page 29.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. **IEEE Trans. Softw. Eng.**, IEEE Press, Piscataway, NJ, USA, v. 8, n. 4, p. 371–379, 1982. ISSN 0098-5589. Citation on page 59.

_____. **Functional program testing and analysis**. New York, NY, USA: McGraw-Hill, Inc., 1986. ISBN 0-070-30550-1. Citation on page 45.

HUO, J.; PETRENKO, A. Transition covering tests for systems with queues. **Softw. Test., Verif. Reliab.**, v. 19, n. 1, p. 55–83, 2009. Citation on page 63.

IEEE. Standard 610.12-1990: Glossary of software engineering terminology. In: _____. **IEEE Standards Software Engineering, Volume One: Customer and Terminology Standards**. [S.l.]: IEEE Press, 1999. v. 1. Citation on page 43.

JAHANIAN, F.; MOK, A. K.-L. Safety analysis of timing properties in real-time systems. **Software Engineering, IEEE Transactions on**, IEEE, n. 9, p. 890–904, 1986. Citation on page 29.

JIA, Y.; HARMAN, M. Higher order mutation testing. **Information and Software Technology**, Elsevier, v. 51, n. 10, p. 1379–1393, 2009. Citation on page 54.

_____. An analysis and survey of the development of mutation testing. **IEEE Trans. Software Eng.**, v. 37, n. 5, p. 649–678, 2011. Citations on pages 22, 54, 55, and 59.

JONES, C. B. **Systematic software development using VDM**. [S.l.]: Prentice Hall Englewood Cliffs, 1990. Citation on page 29.

KATZ, S.; RICHTER, C. A. *et al.* Paris: A system for reusing partially interpreted schemas. In: IEEE COMPUTER SOCIETY PRESS. **Proceedings of the 9th international conference on Software Engineering**. [S.l.], 1987. p. 377–385. Citation on page 29.

KING, J. C. Symbolic execution and program testing. **Commun. ACM**, ACM, New York, NY, USA, v. 19, n. 7, p. 385–394, Jul. 1976. ISSN 0001-0782. Available: <<http://doi.acm.org/10.1145/360248.360252>>. Citation on page 41.

KUHN, D. R. Fault Classes and Error Detection Capability of Specification-Based Testing. **ACM Transactions on Software Engineering and Methodology**, v. 8, n. 4, p. 411–424, 1999. Citations on pages 23, 69, and 77.

LAMSWEERDE, A. v. Formal specification: a roadmap. In: **ACM. Proceedings of the Conference on the Future of Software Engineering**. [S.l.], 2000. p. 147–159. Citations on pages 28 and 29.

LAMSWEERDE, A. V. From system goals to software architecture. In: **Formal Methods for Software Architectures**. [S.l.]: Springer, 2003. p. 25–43. Citation on page 27.

LESTIENNES, G.; GAUDEL, M.-C. Testing processes from formal specifications with inputs, outputs, and datatypes. In: **IEEE Int. Symp.on Software Reliability Engineering, ISSRE**. [S.l.: s.n.], 2002. p. 3–14. Citations on pages 23 and 51.

MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO SIMONE ROCIO SENGHER SOUZA, M. J. M. E. **Introdução ao Teste de Software**. 2004. Nota Didática. Available: <ftp://ftp.icmc.usp.br/pub/BIBLIOTECA/not_did/ND_65.pdf.zip>. Citations on pages 42 and 45.

MEYER, B. On formalism in specifications. **IEEE software**, v. 2, n. 1, p. 6–26, 1985. Citation on page 28.

MILNER, R. **A Calculus of Communicating Systems**. [S.l.]: Springer, 1980. (Lecture Notes in Computer Science, v. 92). ISBN 3-540-10235-3. Citations on pages 30 and 31.

MORELL, L. J. A theory of fault-based testing. **IEEE Transactions on Software Engineering**, IEEE, v. 16, n. 8, p. 844–857, 1990. Citation on page 23.

MORGAN, C. C. **Programming from Specifications**. 2nd. ed. [S.l.]: Prentice-Hall, 1994. Citations on pages 22, 38, and 133.

MOSER, L. E.; RAMAKRISHNA, Y.; KUTTY, G.; MELLIAR-SMITH, P. M.; DILLON, L. K. A graphical environment for the design of concurrent real-time systems. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 6, n. 1, p. 31–79, 1997. Citation on page 29.

MOTA, A.; FARIAS, A.; DIDIER, A.; WOODCOCK, J. Rapid prototyping of a semantically well founded circus model checker. In: **Software Engineering and Formal Methods**. [S.l.]: Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8702). p. 235–249. ISBN 978-3-319-10430-0. Citations on pages 78 and 122.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: SPRINGER. **International conference on Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.], 2008. p. 337–340. Citations on pages 93 and 155.

MYERS, G. J. **The art of software testing**. 2th.. ed. [S.l.]: John Wiley & Sons, Inc., 2004. Citations on pages 43 and 45.

NAUR, P.; RANDELL, B. (Ed.). **Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO**. [S.l.: s.n.], 1969. Citations on pages 28 and 42.

NICOLA, R. D.; HENNESSY, M. C. Testing equivalences for processes. **Theoretical Computer Science**, Elsevier, v. 34, n. 1, p. 83–133, 1984. Citation on page 46.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM, v. 5, n. 2, p. 99–118, 1996. Citation on page 84.

OFFUTT, A. J.; PAN, J. Detecting equivalent mutants and the feasible path problem. In: IEEE. **Computer Assurance, 1996. COMPASS'96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on**. [S.l.], 1996. p. 224–236. Citation on page 58.

OLIVEIRA, M. V. M. **Formal Derivation of State-Rich Reactive Programs Using Circus**. Phd Thesis (PhD Thesis) — University of York, 2006. Citations on pages 39, 65, and 121.

OLIVEIRA, M. V. M.; CAVALCANTI, A. L. C.; WOODCOCK, J. C. P. Unifying Theories in ProofPowerZ. **Formal Aspects of Computing**, online first, 2007. DOI 10.1007/s00165-007-0044-5. Citations on pages 23 and 51.

_____. A UTP Semantics for *Circus*. **Formal Aspects of Computing**, v. 21, n. 1-2, p. 3–32, 2009. Citations on pages 40, 112, and 113.

OLIVEIRA, M. V. M.; SAMPAIO, A. C.; FILHO, M. S. C. Model-checking circus state-rich specifications. In: ALBERT, E.; SEKERINSKI, E. (Ed.). **Integrated Formal Methods**. [S.l.]: Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8739). p. 39–54. ISBN 978-3-319-10180-4. Citations on pages 121 and 123.

PAPADAKIS, M.; MALEVRIS, N. Automatic mutation test case generation via dynamic symbolic execution. In: IEEE. **Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on**. [S.l.], 2010. p. 121–130. Citations on pages 22 and 54.

PAPADAKIS, M.; MALEVRIS, N.; KALLIA, M. Towards automating the generation of mutation tests. In: **Proceedings of the 5th Workshop on Automation of Software Test**. New York, NY, USA: ACM, 2010. (AST '10), p. 111–118. ISBN 978-1-60558-970-1. Available: <<http://doi.acm.org/10.1145/1808266.1808283>>. Citations on pages 83 and 84.

PARASHKEVOV, A. N.; YANTCHEV, J. Arc—a tool for efficient refinement and equivalence checking for csp. In: IEEE. **Algorithms and Architectures for Parallel Processing, 1996. ICAPP'96. 1996 IEEE Second International Conference on**. [S.l.], 1996. p. 68–75. Citation on page 51.

PARNAS, D. L. A technique for software module specification with examples. **Commun. ACM**, ACM, New York, NY, USA, v. 26, n. 1, p. 75–78, Jan. 1983. ISSN 0001-0782. Available: <<http://doi.acm.org/10.1145/357980.358011>>. Citation on page 28.

PAULSON, L. C.; WENZEL, M. **Isabelle/HOL: a proof assistant for higher-order logic**. [S.l.]: Springer, 2002. Citation on page 98.

PELESKA, J.; SIEGEL, M. **Test automation of safety-critical reactive systems**. [S.l.]: Institut für Informatik und Praktische Mathematik, 1996. Citations on pages 46, 48, and 49.

PENA, J.; OLIVEIRA, A. A new algorithm for the reduction of incompletely specified finite state machines. In: **Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on**. [S.l.: s.n.], 1998. p. 482–489. Citation on page [27](#).

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. **J. Object Oriented Program.**, SIGS Publications, Denville, NJ, USA, v. 2, n. 5, p. 13–19, 1990. ISSN 0896-8438. Citation on page [45](#).

PETERSON, J. L. Petri nets. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 9, n. 3, p. 223–252, 1977. ISSN 0360-0300. Citation on page [29](#).

PNUELI, A. The temporal logic of programs. In: IEEE. **Foundations of Computer Science, 18th Annual Symposium on**. [S.l.], 1977. p. 46–57. Citation on page [28](#).

POTTER, B.; TILL, D.; SINCLAIR, J. **An introduction to formal specification and Z**. [S.l.]: Prentice Hall PTR, 1996. Citations on pages [29](#) and [37](#).

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. [S.l.]: McGraw-Hill Professional, 2005. Citations on pages [22](#), [26](#), and [43](#).

RANDELL, B.; BABBAGE, C. The origin of digital computers. In: **Institute of Mathematics and its Applications, November/December**. [S.l.]: SpringerVerlag, 1973. p. 335–346. Citation on page [28](#).

ROSCOE, A. The three platonic models of divergence-strict csp. In: **Theoretical Aspects of Computing-ICTAC 2008**. [S.l.]: Springer, 2008. p. 23–49. Citation on page [47](#).

ROSCOE, A. W. **The Theory and Practice of Concurrency**. [S.l.]: Prentice-Hall, 1998. (Prentice-Hall Series in Computer Science). Citations on pages [22](#), [38](#), and [133](#).

ROSU, G. **K: A Rewriting-Based Framework for Computations—Preliminary version—**. [S.l.], 2007. Citations on pages [95](#), [96](#), [102](#), and [107](#).

ROȘU, G.; ȘERBĂNUȚĂ, T. F. An overview of the K semantic framework. **Journal of Logic and Algebraic Programming**, v. 79, n. 6, p. 397–434, 2010. Citations on pages [96](#) and [155](#).

ROSU, G.; SERBANUTA, T. F. K overview and simple case study. In: **Proceedings of International K Workshop (K'11)**. [S.l.]: Elsevier, 2014. (ENTCS, v. 304), p. 3–56. Citations on pages [107](#) and [154](#).

SAMPAIO, A. C. A.; WOODCOCK, J. C. P.; CAVALCANTI, A. L. C. Refinement in *Circus*. In: ERIKSSON, L.; LINDSAY, P. A. (Ed.). **FME 2002: Formal Methods-Getting IT Right**. [S.l.]: Springer-Verlag, 2002. (Lecture Notes in Computer Science, v. 2391), p. 451–470. Citations on pages [22](#) and [38](#).

SCATTERGOOD, B.; ARMSTRONG, P. **CSPM: A Reference Manual**. [S.l.], 2011. Citations on pages [98](#), [122](#), and [123](#).

SCHORDAN, M. The language of the visitor design pattern. **J. UCS**, v. 12, n. 7, p. 849–867, 2006. Citation on page [85](#).

ȘERBĂNUȚĂ, T. F.; ARUSOAIE, A.; LAZAR, D.; ELLISON, C.; LUCANU, D.; ROȘU, G. **The K Primer (version 3.2)**. [S.l.], 2013. Citations on pages [107](#) and [161](#).

SPIVEY, J. M. **Understanding Z: A formal specification language and its formal semantics.** [S.l.]: Cambridge University Press, 1988. Citations on pages 37 and 162.

_____. **The Z notation: a reference manual.** [S.l.]: Prentice Hall International (UK) Ltd., 1992. Citations on pages 29, 37, 147, 149, 150, and 151.

SRIVATANAKUL, T.; CLARK, J. A.; STEPNEY, S.; POLACK, F. Challenging formal specifications by mutation: a CSP security example. In: **10th Asia-Pacific Software Engineering Conference.** [S.l.]: IEEE Press, 2003. p. 340–350. Citations on pages 23, 69, 70, 74, and 77.

WING, J. M. A specifier's introduction to formal methods. **Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 23, n. 9, p. 8–23, 1990. ISSN 0018-9162. Citation on page 27.

WOODCOCK, J.; LARSEN, P. G.; BICARREGUI, J.; FITZGERALD, J. Formal methods: Practice and experience. **ACM Computing Surveys (CSUR)**, ACM, v. 41, n. 4, p. 19, 2009. Citation on page 26.

WOODCOCK, J. C. P.; CAVALCANTI, A. L. C. A Concurrent Language for Refinement. In: BUTTERFIELD, A.; PAHL, C. (Ed.). **IWFM'01: 5th Irish Workshop in Formal Methods.** Dublin, Ireland: [s.n.], 2001. (BCS Electronic Workshops in Computing). Citations on pages 22 and 38.

WOODCOCK, J. C. P.; CAVALCANTI, A. L. C.; GAUDEL, M.-C.; FREITAS, L. J. S. Operational Semantics for *Circus*. **Formal aspects of computing**, v. 43, p. 100–103, 2007. Citations on pages 23, 40, and 51.

WOODCOCK, J. C. P.; DAVIES, J. **Using Z—Specification, Refinement, and Proof.** [S.l.]: Prentice-Hall, 1996. Citations on pages 22, 38, and 133.

WOODWARD, M. R. Mutation testing—an evolving technique. In: IET. **Software Testing for Critical Systems, IEE Colloquium on.** [S.l.], 1990. p. 3–1. Citation on page 59.

WOODWARD, M. R.; HALEWOOD, K. From weak to strong, dead or alive? an analysis of some mutation testing issues. In: **[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis.** [S.l.: s.n.], 1988. p. 152–158. Citation on page 59.

Z SPECIFICATION EXAMPLE

We present an example to illustrate the way Z specifications are built. The birthday book sample specification was used by [Spivey \(1992\)](#) for a deep presentation of Z, it is simplified here to keep the focus on the features of the language that are relevant for this work. First, the primary constructions of the language are introduced to define the basic data types that are used in the domain of the birthday book application. Two abstract types are declared:

$[NAME, DATE]$

An abstract definition allows the introduction of the required concepts without getting into further details. That leaves freedom for future implementation decisions. A schema that delimits the *state-space* of the system is presented in [Figure 59](#).

The variables of interest are declared above the central division line:

- *people*: the set that should keep the names of the registered people. The special notation ' $\mathbb{P}NAME$ ' is used to specify the domain as the set of all subsets of *NAME* elements;
- *birthday*: function that relates a name to its birthday date.

Figure 59 – Schema *BirthdayBook* in Z

BirthdayBook

people : $\mathbb{P}NAME$

birthday : *NAME* $\rightarrow$ *DATE*

people = dom *birthday*

Functions, as in ‘ $birthday : NAME \rightarrow DATE$ ’, are relations that allows establishing references between elements of two distinct sets. In the example, the function domain is $NAME$ and the function image is $DATE$. There are some variations applicable to functions definitions:

- $X \rightarrow Y$: *total function* defined over all domain elements;
- $X \rightarrowtail Y$: *partial function* defined over some domain elements;
- $X \hookrightarrow Y$: *total injection* defined over all domain elements;
- $X \hookrightarrowtail Y$: *partial injection* defined over some domain elements;
- $X \twoheadrightarrow Y$: *total surjection* over all image and domain elements;
- $X \twoheadrightarrowtail Y$: *partial surjection* over all image elements and some domain elements;
- $X \xrightarrow{\sim} Y$: *total bijection* with each domain element related to a single image element and the other way round;
- $X \rightarrowtail Y$: *partial finite function* that cannot have infinite sets as domain or image;
- $X \hookrightarrowtail Y$: *partial finite injection* that is subject to the previous restrictions and each domain element is related to one and only one image element.

Bellow the central line are the restrictions that applies over the variables. The restrictions must remain valid throughout all the system states, implying that no operation is allowed to break them. In the example, the set *people* should hold the same elements that are present in the domain of function *birthday*. This constraint is a *invariant* of the system. A consequence of the invariant of the example the possibility of getting the whole *people* set from the *birthday* function: we say that *people* is a *derived component*, i.e., a component that can be inferred from parts of other components. This situation yields redundancy and it would be possible to specify the system omitting the *people* set. That economy is not desirable at all: since we are dealing with a specification, it is worth to give names to the real world concepts and keep the model as intelligible as possible.

The schema can also be written in an alternative clear text notation:

BirthdayBook $\hat{=}$
 $[people : \mathbb{P}NAME; birthday : NAME \rightarrowtail DATE \mid people = \text{dom } birthday]$

Values assigned to variables configure a state for the system. For the current example, we might use values like:

$$\begin{aligned}
people &= \{ \text{Foo}, \text{Bar}, \text{Dev} \} \\
birthday &= \{ \text{Foo} \mapsto \text{25-Mar}, \\
&\quad \text{Bar} \mapsto \text{20-Dec}, \\
&\quad \text{Dev} \mapsto \text{20-Dec} \}.
\end{aligned}$$

Given that the three elements of the *people* set are exactly the same elements present in the *birthday* set, the invariant is satisfied. It is worth to notice that, when delimiting the state space, we never established limits for element count in the sets or premature decisions such as date format and other specific details. On the other hand, we have important implicit concepts like: a person should have no more than one birthday date and many people may share a birthday date.

After delimiting the state space, we can define the *operations* that might update the system states. The first operation we define for this example is how to add a new person to the catalog, as shown in [Figure 60](#).

Figure 60 – Operation schema for *AddBirthday*

<i>AddBirthday</i> $\Delta BirthdayBook$ $person? : NAME$ $date? : DATE$
$person? \notin people$ $birthday' = birthday \cup \{person? \mapsto date?\}$

Source: Adapted from [Spivey \(1992\)](#).

The statement $\Delta BirthdayBook$ means that the schema describes a potential state change. Four variables are used: *people*, *birthday*, *people'* and *birthday'*. The first two represent an observation of the state before the application of any changes. The last two variables, decorated with an apostrophe, represent the state after applying the operation changes. Each pair of state variables is restricted to avoid the violation of the invariant $people = \text{dom } birthday$, that should be observed before and after any operation. The operation schema also uses two input parameters: *person?* and *date?*. By convention, input parameters are decorated with an interrogation mark and output parameters with an exclamation mark.

The statements of an operation schema might be split into pre and post conditions. Any statement that involves only plain state variables and input parameters is considered a pre condition. In the example, the *person?* being registered should not already be part of the known

people set. Post conditions should involve decorated state variables and/or output parameters. The *AddBirthday* operation states the extension of the *birthday* function as post condition. The invariant is also a post condition for any state changing operation and, in the example, to extend the *birthday* function implies in changing the *people* derived component, that shall also contain the name added to the function domain $people' = people \cup \{person?\}$. This can be proved:

$$\begin{aligned}
 people' &= \text{dom } birthday' && [\text{invariant before}] \\
 &= \text{dom}(birthday \cup \{person? \mapsto date?\}) && [\text{AddBirthday spec.}] \\
 &= \text{dom } birthday \cup \text{dom } \{person? \mapsto date?\} && [\text{fact on 'dom'}] \\
 &= \text{dom } birthday \cup \{person?\} && [\text{fact on 'dom'}] \\
 &= people \cup \{person?\}. && [\text{invariant after}]
 \end{aligned}$$

To prove properties is one of the strategies for establishing statements about the correctness of what has been described. The mathematical analysis allows the exploration of system behaviour without having an implementation. In the proof presented above, two facts about ‘dom’ come from the following math rule:

$$\begin{aligned}
 \text{dom}(f \cup g) &= (\text{dom } f) \cup (\text{dom } g) \\
 \text{dom}\{a \mapsto b\} &= \{a\}.
 \end{aligned}$$

The operation that retrieves a birthday for a given person is shown in [Figure 61](#).

Figure 61 – Schema for operation *RetrieveBirthday*

<i>RetrieveBirthday</i>
$\exists \text{BirthdayBook}$ $person? : \text{NAME}$ $date! : \text{DATE}$
$person? \in \text{people}$ $date! = \text{birthday}(person?)$

Source: Adapted from [Spivey \(1992\)](#).

The statement $\exists \text{BirthdayBook}$ declares that the operation *RetrieveBirthday* does not apply any changes to the system state. A read-only operation guarantees that an observation of the values in $people'$ and $birthday'$ would show values identical to the values in *people* and *birthday*. The input parameter *person?* is used to receive the name of the person whose birthday

should be retrieved. According to the pre condition $person? \in people$, the operation expects to receive a name that is already registered in the *people* set. The birthday date is returned in the output parameter *date!*.

At last, the schema in [Figure 62](#) contains a more elaborated demonstration of how the set theory may be applied for modelling behaviours. The operations *Reminder* returns all the people that celebrate their birthdays in a given date.

Figure 62 – Schema for the operation *Reminder*

<i>Reminder</i>
$\exists BirthdayBook$
$date? : DATE$
$people! : \mathbb{P}NAME$
$people! = \{ n : people \mid birthday(n) = date? \}$

Source: Adapted from [Spivey \(1992\)](#).

The *Reminder* operation schema imposes no pre conditions, what may cause the output of an empty set in the *people!* parameter when the system state have no registered birthdays for the given date.

The whole birthday book example was presented as a didactic introduction to the basic constructions of the Z notation. For a more comprehensive presentation, the work of [Spivey \(1992\)](#) should be referred.

IMPLEMENTATION OF K-FRAMEWORK REWRITING RULES

B.1 Structural and embedding rules

To handle the text of a *Circus* specification, it is necessary to write rules to process the input and organize it in a way that fits the designed configuration structure. We refer to them as structural rules, and they have neither semantic effect nor counterparts in the operational semantics.

B.1.1 Specification processing and loading

As seen in [subsection 4.3.2](#), the syntax for the source code of a *Circus* specification is a sequence of *Circus* paragraphs, i.e., non-terminals elements of the sort *CircusPar*. We start with a simple rule to enforce the sequential processing of all *CircusPar* that are present in the source code, displayed in (B.1).

$$\text{rule } P1:CircusPar \ P2:CircusPars \Rightarrow P1 \sim> P2 \quad (\text{B.1})$$

The “ $\Rightarrow$ ” symbol is the separator between the term matching and how it should be rewritten. The “ $\sim>$ ” symbol is the sequential list separator for K-computations. After explicitly stating the processing order for the top most structure, we give the rules to deal with the sorts of *CircusPars* expected to be found in a specification. The following rules deal with the declaration of channels.

$$\text{rule } \langle k \rangle \text{ channel } C:Id : Type:Id \Rightarrow . \dots \langle /k \rangle \quad (\text{B.2})$$

$$\begin{aligned} &\langle ch \rangle \dots .Map \Rightarrow C \mid \rightarrow Type \dots \langle /ch \rangle \\ \text{rule } \langle k \rangle \text{ channel } C:Id \Rightarrow . \dots \langle /k \rangle &\quad (\text{B.3}) \\ &\langle ch \rangle \dots .Map \Rightarrow C \mid \rightarrow "sync" \dots \langle /ch \rangle \end{aligned}$$

Note 2 (K-Framework rewrite rules). Rewriting into the K empty symbol, i.e. the “.” dot, means that the matched content will be erased. The triple dots “...”, that may be present either in the beginning or the end of a mentioned cell, means that there may be more content that are not considered in the matching and will remain unchanged after the rewrite step. The “| – >” symbol is the constructor of map pairs.

According to rule (B.2), a paragraph with a channel declaration will be removed from the top of the computation cell after the addition of the name of this channel and its type (for typed channels) to the global configuration cell $\langle ch \rangle$ as a map pair, with the given channel name as key and the channel type as value. When there is no type declared, the channel name is mapped to *sync*, as seen in rule (B.3).

To deal with multiple channel declarations in a single line, we profit from the technique referred as syntax *desugaring* (ROSU; SERBANUTA, 2014), that rewrites a complex sequence of declarations into single-lined simple declarations. This is described in rule (B.4).

```
rule channel S1:SimpleDecl, S2:SimpleDecl, M:MultiDecl => (B.4)
  channel S1 channel S2, M
```

For paragraphs defining named sets, i.e., declaration of uniquely identified sets that holds channel and variable names, we use a strategy that embeds a *Circus* set construction into the representation of sets as used natively in the K-framework. To achieve that, when there is a channel set in the top of the computation cell, we move all its elements, one by one, to the $\langle tempset \rangle$ cell, as seen in rule (B.5). When the set being computed is emptied, it is replaced by the temporary one, which is a pure K-framework set construction, described in rule (B.6).

```
rule <k> { | Xi:Id, Xs:CommaIds | } => { | Xs | } ... </k> (B.5)
  <tempset> ... . => SetItem(Xi) </tempset>
```

```
rule <k> { | .CommaIds | } => TempSet ... </k> (B.6)
  <tempset> TempSet => .Set </tempset>
```

Finally, in rules (B.7) for channels names and (B.8) for variable names, the named set declaration that now utilizes a K-framework set construction is loaded into the global cell $\langle nset \rangle$, holding a map from declared set names to the actual K sets.

```
rule <k> chanset CSName:Id == CSet:Set => . ... </k> (B.7)
  <nset>... .Map => CSName |-> CSet </nset>
```

```
rule <k> nameset NSName:Id == NSet:Set => . ... </k> (B.8)
  <nset>... .Map => NSName |-> NSet </nset>
```

```
rule <k> NSetId:Id => CSet ... </k> (B.9)
  <nset>... NSetId |-> CSet:Set ... </nset>
```

Rule (B.9), reverses the process: when an identifier that refers to a set name is found in the top of the computation cell, that identifier is replaced by the referred set, loaded from the map in $\langle nset \rangle$. The embedding of native K-Framework set type allows the convenient use

of internal operations, such as union, intersection, difference and comparisons. Moreover, the framework provides an interface to SMT solvers, in our case, Z3 (MOURA; BJØRNER, 2008), which is invocable each time a formula involving native types needs to be solved.

Note 3 (Strictness). Here, we rely on the heating and cooling rules of the K-Framework (ROŞU; ŞERBĂNUŢĂ, 2010), i.e., when present in a syntax construction, named sets are marked as strict arguments and pushed to the top of the computation cell (heating). As sets are declared under *KResult* category, the processed native set in the top of the K cell is plugged back to its original syntactical place (cooling).

In order to support more than one process declaration per source text, each paragraph of such syntactical construct is loaded into a structure inside the configuration cell $\langle \text{procstrs} \rangle$. These structures keep references between the declared process name and its textual definition body, named actions, or schemas. This is achieved by rule (B.10), that removes a process declaration from the top of the computation cell and places it into a new $\langle \text{procstr} \rangle$ cell. The process name identifier P is kept into the $\langle \text{pmodel} \rangle$ cell and its textual process definition PD is stored into the $\langle \text{pdef} \rangle$ cell.

```
rule
  <k> process P:Id ^= PD:ProcDef  => . ...</k>
  <procstrs>...
    (.Bag => <procstr>...
      <pmodel> P </pmodel>
      <pdef> PD </pdef>
      ...</procstr>)
  ...</procstrs>
```

(B.10)

To select which process will be processed in the K simulation, we added the command keyword *:run* in the syntax of *CircusPar* (cf. rule (4.6)). As stated in rule (B.11), when the *:run* command is presented with an identifier naming a process model that already exists in some $\langle \text{procstr} \rangle$ structure, the body PD of such process is extracted from the $\langle \text{pdef} \rangle$ to the top of the computation cell, triggering its treatment. The rule also sets the $\langle \text{model} \rangle$ configuration cell of the current *proc* environment to the name of the loaded model.

```
rule <k> :run ProcId:Id => PD ... </k>
  <model> _ => ProcId </model>
  <procstr>...
    <pmodel> ProcId </pmodel>
    <pdef> PD </pdef>
  ...</procstr>
```

(B.11)

Once loaded in the configuration cells, the next step is to deal with the presence or absence of process paragraphs in the header of the process (cf. rules (4.13) and (4.14)). Such paragraphs may declare a state, some named actions and $\mathbb{Z}$ schemas before the main action. Rule

(B.12) deals with a process that contains only the main anonymous action, moving this action straight to the top of the computation cell and discarding the rest of the construct.

```
rule <k> begin @ Pa:ParAction end => Pa ... </k> (B.12)
```

Rules (B.13) and (B.14) deal with processes with process paragraphs. Such rules simply establish the sequential processing of the *PPars* before the main action (B.13) and enforces the natural processing order of each process paragraph (B.14).

```
rule <k> begin P:PPars @ A:ParAction end => P ~> A ... </k> (B.13)
```

```
rule P1:PPar P2:PPars => P1 ~> P2 (B.14)
```

Rule (B.15) deals with the declaration of a named action in a process paragraph. It removes the declaration from the computation cell and inserts the action body into the *< pact >* map of the correct *< procstr >* structure, using the model of the process and the name of the action as references.

Similarly, rule (B.16) describes how the declared $\mathbb{Z}$ schemas are processed and loaded.

```
rule (B.15)
```

```
<k>N:Id ^= A:Action => . ... </k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pact> LMap => LMap[N <- A] </pact>
...</procstr>
```

```
rule (B.16)
```

```
<k> N:Id == S:SchemaExp => . ... </k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <psch> LMap => LMap[N <- S] </psch>
...</procstr>
```

Consequently, a rule is needed that takes the named action definition from its storage and places its body into the top of the computation cell: rule (B.17) is triggered by the occurrence of a named action, i.e., the next computation is an identifier that refers to a valid named action. Valid means that it exists in the named action map *< pact >* under the *< procstr >* configuration of the model *PModel*, from the process under treatment.

Rule (B.18) that fetches a schema by its name is similar to the one for actions, but it also requires the preparation of a new environment for treating the decorated $\mathbb{Z}$ identifiers. The conditions in an operation schema, i.e., schemas that may change state or local variables, can involve variable assignments from both before and after the schema application. For that reason, the current environment *env* is replicated into the *postenv* cell, and this replica can be manipulated and compared with the original *env*. If the schema condition is satisfied, the replica replaces the original copy.

rule (B.17)

```
<k> A:Id => (B) ...</k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pact>... A |-> B ...</pact>
...</procstr>
```

rule (B.18)

```
<k> A:Id => B ...</k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <psch>... A |-> B ...</psch>
...</procstr>
<env> Env </env>
<postenv> _ => Env </postenv>
```

B.1.2 Configuration maintenance

The rules presented in this subsection are designed to maintain the structures of configuration cells, performing optimizations, and cleaning those pieces that are not required anymore, such as discarded processes and their initials.

Rule (B.19) is designed to delete an *ok* notification structure when there are no more process references in the addressing set *to*. This is accomplished by matching the empty set (*.Set*) and rewriting the whole bag of cells to the empty cell bag (*.Bag*).

rule (B.19)

```
<oks>
  (<ok>... <to>.Set</to> ...</ok> => .Bag)
...</oks>
```

Another rule, (B.20), deletes an *init* structure that became useless due to the discarding of all the processes referred in the *from* set. To check such property, we require the subset inclusion $From \subset Disc$, where *Disc* is the set of all discarded processes, registered in the *discarded* global configuration cell.

rule (B.20)

```
<discarded>Disc</discarded>
<inits>...
  (<init>... <from> From </from> ...</init> => .Bag)
...</inits>
requires (From <=Set Disc)
```

The rule that removes the *proc* environments of discarded processes is shown in (B.21). It requires the matching of the process reference *PI* with an element of the set inside the *discarded* cell.

```

rule
  <discarded> ... SetItem(PId) ... </discarded>
  <procs>...
    <proc>...
      <id>PId</id>
      ...</proc> => .Bag
    ...</procs>

```

(B.21)

It is possible that a discarded process leave children environments. For such cases, rule (B.22) performs the removal of these orphans *proc* environment and add the reference of the removed orphan to the set of discarded processes.

```

rule
  <discarded>
    Disc SetItem(PId:Int) => SetItem(PId) Disc SetItem(Orphan)
  </discarded>
  <procs>...
    <proc>...
      <id>Orphan:Int</id>
      <parent>PId</parent>
      ...</proc> => .Bag
    ...</procs>

```

(B.22)

B.2 Operational semantics rules

Once the configuration is loaded and an action is present in the top of the computation cell, the rewriting engine is expected to proceed accordingly to the semantics of *Circus*: it is mandatory to keep the rewrite rules faithful to the formal operational semantics. Our implementation is based on such rules as given in (CAVALCANTI; GAUDEL, 2013) and recalled in Appendix C.1.

As usual, the operational semantics of *Circus*, abbreviated to OPS in the sequel, relies upon configurations and transition rules between configurations. As seen in section 2.3, configuration consists of three parts: a logical constraint between some symbolic variables, a state/environment definition mapping *Circus* variables into symbolic variables, and a continuation, i.e. the remaining *Circus* action. The application of an OPS rule will modify at least one of these parts. An OPS rule is defined by: a precondition, i.e. a logical expression on symbolic variables; an observable or an empty label, noted ε ; and two configuration patterns, for the configurations before and after the rule application.

Transposed into the K-Framework, i. e. in the configuration structure presented in subsection 4.3.3, the constraint is a logical expression in the *const* cell, the state/environment is a map in the *env* cell and the remaining action is processed at the *k* computation cell. The rewrite rules take into consideration the current configuration cells and, if the logical precondition applies, the content of the cells is rewritten according to the patterns.

The rules presented in this section are paired with some reference to the original OPS rules that motivated them. Ideally, we would have exactly one rewrite rule in K for each OPS rule. However, due to particularities of the K-framework and the complexity of the semantics of *Circus*, this was not always feasible. In such cases, we report the reasons that motivated our decisions.

B.2.1 Internal progress

We present here the implementation of those OPS rules that does not produce observable labels, i.e., rules that guides the progress of processes while producing an ε label and, consequently, does not require any appending to the symbolic trace *tr* cell. Nevertheless, they may induce the appending of specification labels to the specification trace contained in the *spectr* cell.

The assignment of variables is described by the *Circus* OPS rule C.1.2. There is no precondition. A new fresh symbolic value ($?W$) is assigned to the given variable identifier V and a condition ($?W == E$) is appended to the constraint. This condition imposes an equality relation between the new symbolic value and the expression (E) given in the right hand side of the assignment construct. As the syntax of variable assignment is tagged for strictness in its right side element, the value of the expression is evaluated before the treatment of the assignment itself. After this treatment, the action in the new configuration gets a *Skip* in place of the treated assignment.

In our K-implementation of the OPS, there are two rewrite rules to accomplish this semantic behavior, due to the fact that we add the possibility to have invariants in state definitions. This leads to treat differently local and state variables. The first case is covered by rule (B.23), where the explicit requirement in (B.24) guarantees that this rule will only be applied for a variable V that is not a name in the set of keys from the state map St .

Dealing with such invariants in K is straightforward and they allow more concise and abstract specifications.

rule (B.23)

```

<k> V:Id := E:Int => Skip ... </k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pstate> St </pstate>
...</procstr>
<env> Env => Env[V <- ?W:Int] </env>
<const> Const => Const andBool (?W ==Int E) </const>
<spectr>... . => V := E </spectr>
requires (notBool (V in (keys(St))))

```

(B.24)

The assignment to state variables is treated by rule (B.25). This separation is motivated by the need of keeping the validity of the state invariant after changing the value of a state

variable. After reassigning a symbolic value to a state variable, it is necessary to rewrite the state invariant into the constraint. This is simply achieved by adding these conditions and, relying on SMT optimizations, to remove the old clauses that are no more required in the configuration. To trigger this extra step, we insert the construction `state Inv ~> Skip ...` in the resulting action before the `Skip` originally required by the OPS rule C.1.2. The computation triggered by this auxiliary construction is further described within the rules concerning state evaluations (rules (B.35) and (B.34)).

rule (B.25)

```

<k> V := E:Int => state Inv ~> Skip ... </k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pstate> ... V |-> _ ... </pstate>
  <pinv> Inv:BoolExp </pinv>
...</procstr>
<env>... V |-> (_ => ?W:Int) ...</env>
<const> Const => Const andBool (?W ==Int E) </const>
<spectr>... . => V := E </spectr>

```

The *declaration and scope handling for local variables* in *Circus* are described by OPS rule C.1.6, rule C.1.7 and rule C.1.8. In *Circus*, the precondition for a variable declaration concerns its type and name. As the current implementation does not handle type checking, the declared name is considered only. This precondition imposes $V \notin \alpha s$, i.e., the given name V for a variable is not already used in the alphabet αs . Since the parser already rejects identifiers that conflicts with syntactical tokens, the condition of the rule we introduce in (B.26) only checks if V is not a name of a state variable. The *Circus* OPS also requires every declared variable to be assigned to a fresh symbolic value, reflected in K by the use of the fresh identifier generator $?W$.

rule (B.26)

```

<k> var V:Id @ A:Action => A ~> end V ...</k>
<env> Env => Env[V <- ?W:Int] </env>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pstate> St </pstate>
...</procstr>
<stack> . => ListItem(Env) ... </stack>
requires (notBool (V in (keys(St))))

```

OPS rule C.1.6 introduces the keyword `let`, to handle the scope of the declared variable in the remaining action. In our K-Framework implementation, we have considered that it is not necessary to use such keyword since the declared variables names are available in the environment *env* cell. As defined by the original OPS rule, in rule (B.26) we state that the

declared variable scope is restricted to the action A specified after the @ symbol. This is ensured by appending “end V” to be evaluated after the action A.

To guarantee the possibility of reusing variable names in nested scopes, a stack of environments is used and the current *env* configuration is pushed onto the *stack* cell at each new variable declaration. Once a scope ends, the previous environment is popped back into the *env* cell. The assignments of variables declared in the ending scope are lost. All the other assignments are kept by the *updateMap* operation, as shown in rule (B.27).

```
rule
  <k> end V:Id => . ... </k>
  <env>
    RetEnv => updateMap(Env, removeAll(RetEnv, SetItem(V)))
  </env>
  <stack> ListItem(Env) => .List ... </stack>
  <spectr>... . => end V </spectr>
```

(B.27)

For the *sequential composition of processes*, the *Circus OPS* includes the rule C.1.9 and rule C.1.10. The first defines the continuation of an action A1 in the left hand side of the operator while there are computations to be performed. The OPS rule C.1.10 defines the continuation with the right hand side action A2 once the computations in A1 finishes with a Skip. In our implementation, the described behavior is achieved by ordering the computations of two sequentially composed actions, as presented in (B.28). To guarantee the correct continuation after reaching Skip in the first executed action, we include rule (B.29).

```
rule
  <k> A1:Action ; A2:Action => A1 ~> A2 ... </k>
```

(B.28)

```
rule
  <k> Skip ~> A:Action => A ... </k>
```

(B.29)

The behavior of the *internal choice* construction is given in *Circus OPS* rule C.1.11, consisting in two parts: the process unconditionally behaves as the left hand side action or the right hand side action of the operator. This choice is made non-deterministically and, for that reason, the rules in (B.30) and (B.31) are tagged with the [transition] annotation. When a transition is specified, the rewriting engine marks the current step for backtracking when searching for alternative executions (ȘERBĂNUȚA *et al.*, 2013). The evaluation and output of all possible outcomes is triggered by the switch `-search` when invoking the *krun* tool.

```
rule
  <k> A1:Action |-| A2:Action => A1 ... </k> [transition]
```

(B.30)

```
rule
  <k> A1:Action |-| A2:Action => A2 ... </k> [transition]
```

(B.31)

The *guarded actions* are defined in *Circus OPS* rule C.1.12. It states that the computation of an action A guarded by a logical condition G results in continuing the process with the action A

while adding the condition specified in G to the constraint. The precondition for applying the rule requires that the constraint remains valid after adding G . Rule (B.32) strictly reproduces such behavior.

Remark

[Guarded actions in specification traces] To write the specification text of a guard in a specification trace, we use an intermediary step that is not given here. It is available in the source code listing of the implementation. $\square$

```
rule (B.32)
  <k> G:Bool & A:Action => A ... </k>
  <const> Const => Const andBool G </const>
  requires Const andBool G
```

B.2.2 State and Z schema

Due to its richness and its complexity, the semantics of $\mathbb{Z}$ (SPIVEY, 1988) would deserve to be treated in a full companion project. In absence of such a project, we decided to give semantics to a subset of $\mathbb{Z}$ schema constructs in the K-Framework, allowing the declaration and handling of states for *Circus* processes. In a nutshell, we deal with $\mathbb{Z}$ constructs where variables and formulas are written using the K predefined types and operators for integers and booleans.

We divide the schemas into two types: the state schemas and operation schemas. A process declaration can include a single state schema only and may contain multiple operation schemas. As seen for the assignment of variables (rule B.25) state schemas contain declarations of state variables and possibly an invariant. This is an addition to the OPS.

Operation schemas define state changes via pre- and post-conditions. These are logical expressions on state variables and input/output schema variables (that may reference any variable name within the current environment). Variables are possibly decorated in the post-condition: they describe the effect of the operation on these variables.

The variables declared in the state schema are treated by rule (B.33), processing each name V in the list of multiple declarations M . The variable names and types are inserted both into the *state* and *env* maps. Since all *Circus* variables are arbitrarily initialised, a fresh symbolic value $?W$ is assigned to each declared variable.

Once there are no more variable declarations to be processed in the declaration part of the schema, rule (B.34) is applied, storing the state invariant expression B into the *inv* cell and scheduling the evaluation of the invariant for the next computation step.

rule (B.33)

```

<k> state [ (V:Id , M:MultiDecl => M) | B:BoolExp ] ...</k>
<env>... .Map => V |-> ?W:Int ...</env>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pstate>... .Map => V |-> "int" ...</pstate>    <pinv> Inv:BoolExp </pinv>
...</procstr>

```

rule (B.34)

```

<k> state [ .MultiDecl | B:BoolExp ] => state B ...</k>
<model> PModel:Id </model>
<procstr>...
  <pmodel> PModel </pmodel>
  <pinv> _ => B </pinv>    ...</procstr>

```

Rule (B.35) triggers the evaluation of the state invariant using the current *env* variable assignment. If the invariant is valid within the context of the constraint in *const*, the rule is applied, the invariant expression is added to the constraint and the process continues. If the invariant is invalidated, no rule applies.

rule (B.35)

```

<k> state B:Bool => . ...</k>
<const> Const => Const andBool B </const>
requires Const andBool B

```

Note that rule (B.25) forces the computation of the invariant after the treatment of assignments to state variables, rule (B.34) here for operations schema.

The behavior induced by a reference to an operation schema inside a *Circus* action is given by the OPS rule C.1.1. In our implementation, once ready for computation, the schema is loaded according to the structural rule (B.18). Then, rewrite rule (B.36) begins a sequence of steps by processing the declaration part: each variable gets a fresh symbolic value into its *postenv* cell map entry. This is necessary for handling decorated identifiers in logical expressions involved in post-conditions.

rule (B.36)

```

<k> [ (V:Id , M:MultiDecl => M) | B:BoolExp ] ...</k>
<postenv> Postenv => Postenv[V <- ?W:Int] </postenv>

```

Similarly as ensured by the rewrite rule (B.34), when the variable declaration part of an operation schema is emptied, rule (B.37) takes place and puts the schema logical expression B to be processed in the <k> cell. The computation is triggered by the internal keyword *schema* followed by the logical expression.

rule (B.37)
`<k> [ .MultiDecl | B:BoolExp ] => schema B...</k>`

rule (B.38)
`<k> schema B => state Inv ~> Skip ...</k>`
`<const> Const => Const andBool B </const>`
`<env> _ => PostEnv </env>`
`<postenv> PostEnv => .Map </postenv>`
`<model> PModel:Id </model>`
`<procstr>...`
`<pmodel> PModel </pmodel>`
`<pinv> Inv </pinv> ...</procstr>`
`requires Const andBool B`

Rule (B.38) defines the evaluation of the schema logical expression B in the context of the current environment env and the post-condition environment in $postenv$. If the conjunction of the condition with the current constraint $const$ is valid, the rule is applied and all the involved configuration elements are updated. The OPS rule C.1.3 of *Circus* operational semantics defines *Skip* as the continuation of a successful operation schema application. In rule (B.38), we force the reevaluation of the state invariant before *Skip*.

B.2.3 Observable progress

We present the rules that model the synchronization strategy described in subsection 4.3.4. Such rules are designed to post, forward and resolve the *init* structures that represents the potential initials after an action.

We refrain from presenting all the rules when subtle structural differences are necessary to handle initials for events, inputs and outputs. In such cases, only the rule for events is presented and further details can be obtained consulting the complete listing of rules.

Initial posting

The posting of an initial is necessary when the current computation involves a communication or an event. Rule (B.39) deals with the case of an event, following the *Circus* OPS rule C.1.4 for outputs, just omitting the value to be outputted. (actually, there is no specific rule for the case of event synchronisation in the version of the *Circus* operational semantics we use).

When an identifier C referring to a channel name is in the top of the computation cell, i.e., an event on channel C is required, the *sync* flag is lifted and an initial is created inside the bag *inits*. This new initial refers in cell *iresp* to the *proc* environment currently responsible for it, namely PId . It keeps in *iev* the name of the potential event, namely C . The set of subscriber processes is in *from*, initialized by the reference to the current *proc* environment, and the set of references to alternatives, in *ialt*, is initialized with the empty set.

rule (B.39)

```

  <k> C:Id ... </k>
  <sync> false => true </sync>
  <id>PId:Int</id>
  <ch> ... C |-> _ ... </ch>
  <inits>...
  .Bag => <init>
    <iresp>PId</iresp>
    <iev>C</iev>
    <from>SetItem(PId)</from>
    <ialt> .Set </ialt>
  </init>
</inits>
[transition]

```

The transition annotation is used because the order of unsynchronized communications is non-deterministic.

After a rule for posting the initial, we present the forwarding rules sketched in [subsection 4.3.4](#), first for channel hiding, then for parallel composition, and finally for external choice.

Channel hiding

The behavior induced by hiding observable communications of some channels is defined in *Circus* OPS rules [C.1.26](#), [C.1.27](#) and [C.1.28](#). To correctly reproduce this behavior, we use the tree structure presented in [subsection 4.3.4](#): when a channel hiding operator is applied to an action A, a child *proc* environment is spawned to continue action A.

As shown in rule (B.40), the new environment gets a copy of all the semantically relevant configuration cells: *env*, *model* and *const*; a fresh integer reference is generated by incrementing the integer inside the global *nextid* cell; the reference to the parent *proc* environment is kept in the cell *parent*. The parent computation cell is rewritten into the internal construction *hide* with two parameters: the child *proc* environment reference (*L* +Int 1) and the set CS of channels to be hidden.

The forwarding of all the initials posted by the action in the child *proc* environment will necessarily pass through the responsibility of the parent environment for reaching upper levels and, if succeeding to reach the top level, being appended to the traces. The hierarchical structure allows the design of rewrite rules to filter the initials on their way to the top level according to the OPS rules [C.1.26](#) and [C.1.27](#).

The case covered by OPS rule [C.1.26](#) is the occurrence of an event or communication that does not involve a hidden channel, i.e., a channel that is not an element of the hiding channel set CS. As shown in rewrite rule (B.41), the forwarding is performed by replacing the *iresp* reference of the matched *init* from the child CId to the parent PId.

rule (B.40)

```

<k>A:Action /H CS:Set
  => hide (L +Int 1) CS ... </k>
<nextid> L => L +Int 1 </nextid>
<id> ParId </id>
<model> PModel:Id </model>
<env> Env </env>
<const> Const </const>
(.Bag =>
  <proc>...
    <k> A </k>
    <model> PModel </model>
    <parent> ParId </parent>
    <id> (L +Int 1) </id>
    <alt> (L +Int 1) </alt>
    <env> Env </env>
    <const> Const </const>
  ...</proc>)

```

rule (B.41)

```

<proc>...
  <k> hide CId:Int CS:Set ... </k>
  <id> PId:Int </id>
...</proc>
<inits>...
  <init>...
    <iresp>CId => PId</iresp>
    <iev>C:Id</iev>
  ...</init>
... </inits>
requires notBool (C in CS)

```

OPS rule C.1.27 covers the case where the involved channel is a member of the hiding set. No observable label is produced, but the evaluation of the action proceeds. Thus, in rule (B.42) the initial is resolved before reaching the top level environment, therefore canceling its potential writing into the traces. This removes the matched *init* cell and spawns the corresponding *ok* notification, allowing the action to continue as if the initial was actually written into the traces. The rewrite rule contains some additional conditions to avoid posting *ok* notifications to discarded processes.

The finalization of a hidden action is defined in OPS rule C.1.28: when the hidden action reaches a *Skip*, this closes the scope of the hiding. Thus the hiding operator and channel set are removed from the action text, which is reduced to a single *Skip*. This is ensured by rule (B.43), where the semantically relevant configuration cells are copied from the child environment under finalization back to the parent for continuing the remaining evaluations.

rule (B.42)

```

<proc>...
  <k>  hide CId:Int CS:Set  ... </k>
  <id> PId:Int </id>
...</proc>
<inits>
  (<init>...
    <iresp>CId</iresp>
    <ialt>Alt</ialt>
    <iev>C:Id</iev>
    <from>FSet</from>
    </init> => .Bag)
... </inits>
<oks>...
  (.Bag =>
    <ok> <oev>C</oev> <to>FSet -Set Disc</to> </ok>)
... </oks>
<discarded>Disc => Disc Alt</discarded>
requires notBool (size(FSet -Set Disc) ==Int 0)
      andBool (C in CS)

```

rule (B.43)

```

<k>hide CId:Int CSet:Set => Skip ... </k>
<id> ParId </id>
<env> _ => Env </env>
<const> _ => Const </const>
(<proc>...
  <k> Skip </k>
  <id>CId</id>
  <parent> ParId </parent>
  <env> Env </env>
  <const> Const </const>
...</proc> => .Bag)

```

Parallel composition

The parallel composition operator is defined by the *Circus* OPS rules C.1.20 to C.1.25. The evaluation of such a structure begins with the spawning of two parallel actions A1 and A2 inside new children *proc* environments: each one will correspond to the left and the right parts of the composition.

We designed the rewrite rule (B.44) to match OPS rule C.1.20. The context of each child *proc* environment replicates the semantically relevant configuration cells from the parent *proc*. The computation cell of the parent is rewritten, starting with the internal keyword *par*, followed by: the fresh references to the left ($L + \text{Int } 1$) and right ($L + \text{Int } 2$) child *proc* environments, the set *CSet* of synchronization channels, two disjoint sets *X1* and *X2* of variable names to handle the parallel assignment of variables.

rule (B.44)

```

<k>
  A1:Action [|X1:Set|CSet:Set|X2:Set|] A2:Action =>
  par ListItem(L +Int 1) ListItem(L +Int 2), CSet,X1,X2
... </k>
<nextid> L => L +Int 2 </nextid>
<id> ParId </id>
<model> PModel:Id </model>
<env> Env </env>
(.Bag =>
  <proc>...
    <k> A1 </k>
    <model> PModel </model>
    <parent> ParId </parent>
    <id> (L +Int 1) </id>
    <alt> (L +Int 2) </alt>
    <env> Env </env>
  ...</proc>
  <proc>...
    <k> A2 </k>
    <model> PModel </model>
    <parent> ParId </parent>
    <id> (L +Int 2) </id>
    <alt> (L +Int 1) </alt>
    <env> Env </env>
  ...</proc>)

```

We also provide two complementary structural rules: rule (B.45) accepts the syntax for parallel composition with no parallel state writing; and rule (B.46) transforms the syntax of an interleaving in a parallel composition with no synchronizations.

rule (B.45)

```

<k>
  A1:Action [|CS:Set|] A2:Action
=> A1:Action [| .Set|CS|.Set|] A2:Action
... </k> [structural]

```

rule (B.46)

```

<k>
  A1:Action ||| A2:Action
=> A1:Action [| .Set|] A2:Action
... </k> [structural]

```

Once spawned, both actions are evaluated independently until the posting of an initial occurs. Forwarding of initials inside a parallel composition takes into consideration the synchronization of the concurrent actions when there are events or communications over channels in the CSet synchronization set.

Initials involving a channel name C that is not in $CSet$ are forwarded according to the OPS rules C.1.22 and C.1.23, i.e., they are forwarded immediately. Since we do not discriminate which side of the parallel composition is treated, only one rewrite rule is necessary for defining the behavior of both OPS rules. Rule (B.47) matches a *proc* environment CId that is a child of a parallel composition environment PId and has an initial that respects the condition `notBool (C in CSet)`. Once matched, the rule forwards it, rewriting the responsible *iresp* from the child to the parent ($CId \Rightarrow PId$).

rule (B.47)

```

<proc>...
  <id>CId:Int</id>
  <parent>PId:Int</parent>
...</proc>
<proc>...
  <id>PId</id>
  <k> par _:List,CSet:Set, _:Set, _:Set ... </k>
...</proc>
<inits>...
  <init>...
    <iresp>CId => PId</iresp>
    <iev>C:Id ...</iev>
  ...</init>
... </inits>
requires notBool (C in CSet)

```

For initials over channel names in the synchronization set, the forwarding is postponed until there is another initial, posted by the environment at the opposite side of the parallel composition, configuring one of the synchronization behaviors as defined in OPS rules C.1.24 or C.1.25.

Rule (B.48) handles the forwarding of two synchronized initials: it matches two children environment, the left side LId and the right side RId ; a parent parallel composition *proc* environment PId ; and two initials, one for each child *proc*. The two initials involved in a synchronization are merged into a single one that will carry in its *from* and *alt* sets all the references to the standing by environments from both. The initial carrying all the references is forwarded and the other is removed.

An input/output synchronization, as defined in OPS rule C.1.24, establishes the communication of a value between the two sides of a parallel composition. Such cases are treated by rule (B.49). We omit the *proc* environment matching part, since it is identical to rule (B.48). The difference is the explicit requirement of matching an input value with an output value over the same synchronized channel. The initials are merged within the output communication, since the output label is the correct observable behavior accordingly to the *Circus* OPS.

The finalization of a parallel composition is given in the OPS rule C.1.21. When both

rule (B.48)

```

<proc>...
  <id>LId:Int</id>
  <parent>PIId:Int</parent>
...</proc>
<proc>...
  <id>RId:Int</id>
  <parent>PIId:Int</parent>
...</proc>
<proc>...
  <id>PIId</id>
  <k> par _:List,CSet:Set, _:Set, _:Set ... </k>
...</proc>
<inits>...
  <init>...
    <iresp>LId => PIId</iresp>
    <iev>C:Id ...</iev>
    <ialt> LAlt => LAlt RAlt </ialt>
    <from> LFrom => LFrom RFrom </from>
...</init>
  (<init>...
    <iresp>RId</iresp>
    <iev>C ...</iev>
    <ialt>RAlt</ialt>
    <from>RFrom</from>
    ... </init> => .Bag)
... </inits>
requires (C in CSet)

```

rule (B.49)

```

(...)
<inits>...
  <init>...
    <iresp>LId => PIId</iresp>
    <iev>C:Id ! E:Exp</iev>
    <ialt> LAlt => LAlt RAlt </ialt>
    <from> LFrom => LFrom RFrom </from>
...</init>
  (<init>...
    <iresp>RId</iresp>
    <iev>C:Id ? _</iev>
    <ialt>RAlt</ialt>
    <from>RFrom</from>
    ... </init> => .Bag)
... </inits>
requires (C in CSet)

```

parallel actions reach their final Skip, the whole parallel composition continues as a single Skip and the variable assignments performed in the parallel branches are merged accordingly to the

name sets $X1$ and $X2$. Only assignments made in the left (resp. right) side to variable names listed in $X1$ (resp. $X2$) are kept.

Rule (B.50) performs this step, removing the two finished children environments and merging both assignment maps in the *env* cells into the parent *proc* environment, taking into consideration the name sets $X1$ and $X2$.

rule (B.50)

```

(<proc>...
  <id>LId:Int</id>
  <k> Skip </k>
  <parent>PId:Int</parent>
  <const> LConst </const>
  <env> LEnv </env>
...</proc> => .Bag )
(<proc>...
  <id>RId:Int</id>
  <k> Skip </k>
  <parent>PId:Int</parent>
  <const> RConst </const>
  <env> REnv </env>
...</proc> => .Bag )
<proc>...
  <id>PId</id>
  <const> Const =>
    Const andBool LConst andBool RConst
  </const>
  <env> Env =>
    updateMap(Env,
      removeAll(LEnv, keys(LEnv) -Set X1)
      removeAll(REnv, keys(REnv) -Set X2)
    )
  </env>
  <k> par ListItem(LId) ListItem(RId),
    CSet:Set, X1:Set, X2:Set => Skip ...</k>
...</proc>

```

External choice

Starting an external choice, as defined in the OPS rule C.1.13, is similar to starting a parallel composition: the *Circus* process is split into two actions and, as given in OPS rules C.1.16 and C.1.17, they can proceed until an initial is posted. Rule (B.51) implements the behavior of OPS rule C.1.13, spawning two *proc* environments to evaluate each action, and rewriting the computation cell into the internal keyword choice followed by a set of two elements containing the fresh references to the spawned environments. Actually, the *Circus* OPS rules C.1.16 and

[C.1.17](#) do not require an explicit rewrite rule, since they deal with internal progress, and internal progress does not involves initial handling as seen in [subsection B.2.1](#)

rule (B.51)

```

<k> A1:Action [] A2:Action =>
  choice SetItem(L +Int 1) SetItem(L +Int 2) ... </k>
<nextid> L => L +Int 2 </nextid>
<id>PId:Id</id>
<model> PModel:Id </model>
<env> Env </env>
<const> Const </const>
(.Bag =>
  <proc>...
    <k> A1 </k>
    <model> PModel </model>
    <parent>PId</parent>
    <env> Env </env>
    <const> Const </const>
    <id> (L +Int 1) </id>
    <alt> (L +Int 2) </alt>
  ...</proc>
  <proc>...
    <k> A2 </k>
    <model> PModel </model>
    <env> Env </env>
    <const> Const </const>
    <id> (L +Int 2) </id>
    <alt> (L +Int 1) </alt>
    <parent>PId</parent>
  ...</proc>)

```

As stated by OPS rule [C.1.18](#) and rule [C.1.19](#), once an observable event or communication is performable at one of sides of the external choice, the other side is no more considered. The *Circus* process continues with the observable event or communication and its continuation. To achieve such a behavior, the forwarding of initials inside an external choice is handled by rule [\(B.52\)](#). Initials are forwarded as soon as they are posted by one of the sides. For each forwarded initial, we mark in the set *alt* of alternatives the reference to the *proc* environment in the other side of the choice. When an initial with a non-empty set of alternatives is resolved, all *proc* environments referred in the set are discarded and only the chosen side of each external choice gets an *ok* notification to proceed.

The finalization of an external choice is given by the OPS rules [C.1.14](#) and [C.1.15](#). When one of the choice sides reaches its final *Skip*, the external choice ends, and the *Circus* process continues with the assignments and constraint from the successfully finished side. By using the transition annotation, we capture the behavior of both OPS rules with rule [\(B.53\)](#).

rule (B.52)

```

<proc>...
  <id>PId</id>
  <k> choice Set ... </k>
...</proc>
<proc>...
  <id>CId</id>
  <parent>PId</parent>
  <alt>AId</alt>
...</proc>
<inits>...
  <init>...
    <iresp>CId => PId</iresp>
    <ialt>... .Set => SetItem(AId) ...</ialt>
  ... </init>
... </inits>

```

rule (B.53)

```

<procs>...
  <proc>...
    <id>PId:Int</id>
    <k> choice _ => Skip ... </k>
    <env> _ => Env </env>
    <const> _ => Const </const>
  ...</proc>
  (<proc> ...
    <id>P1:Int</id>
    <parent>PId</parent>
    <alt>AId:Int</alt>
    <k>Skip</k>
    <env> Env </env>
    <const> Const </const>
  ...</proc> => .Bag)
...</procs>
<discarded>...
  .Set => SetItem(AId)
...</discarded>
[transition]

```

Initial resolution

As explained in [subsection 4.3.4](#), the resolution of an initial is the end of its life-cycle. It is removed from the set of initials and two consequences may follow: the appending of its content to the traces and the generation of an *ok* notification to allow the continuation of *proc* environments affected by this initial.

The resolution rewrite rules are triggered in two cases. The first one is when an initial

is embedded in the scope of a hiding operation. This case is treated in rule (B.42). The other one is when an initial has reached the top level, i.e. its *iresp* sub-cell references the *proc* Id 0 environment. It is treated in rule (B.54), which ensures the removal of the *init*, the appending of the event *iev* to the trace *tr* and specification trace *spectr* and, finally, the spawning of the *ok* notification, addressed to all the *proc* environments listed in the set *from* of the resolving *init*. Extra precautions are taken to avoid the resolution of an initial that concerns only discarded processes, such cases are treated by the structural rule in (B.20).

rule (B.54)

```

<id>0</id>
(<init>...
  <iresp>0</iresp>
  <ialt>Alt</ialt>
  <iev>C:Id</iev>
  <from>FSet</from>
</init> => .Bag)
<oks>...
  (.Bag => <ok>
    <oev>C</oev>
    <to>FSet -Set Disc</to>
  </ok>)
... </oks>
<tr>... . => C </tr>
<spectr>... . => C </spectr>
<discarded>Disc => Disc Alt</discarded>
requires notBool (size(FSet -Set Disc) ==Int 0)

```

When an *ok* notification is available and addressed to a standing-by *proc* environment, i.e., holding true in its *sync* flag sub-cell, rule (B.55) is triggered. It matches such standing-by *proc* PId referenced in an *ok* structure and its event *oev* with the *proc* current computation. As result, the current computation is concluded and the *sync* flag is set to false, allowing the continuation of the remaining computations in *proc*. The reference PId is also removed from the *to* set of the *ok* notification. The notification itself remains available to match possible other *proc* environments in the remaining references. When the *to* set becomes empty, the structural rule in (B.19) removes the *ok* structure from the configuration.

rule (B.55)

```

<k> C:Id => . ... </k>
<sync> true => false </sync>
<id>PId:Int</id>
<oks>...
  <ok>
    <oev>C:Id</oev>
    <to>... SetItem(PId) => .Set ...</to>
  </ok>
... </oks>

```


CIRCUS OPERATIONAL SEMANTICS

C.1 Operational semantics rules

$$\frac{c \wedge (s; p) \wedge (\exists v' \bullet s; Q)}{(c \mid s \models p \vdash Q) \xrightarrow{\varepsilon} (c \wedge (s; Q[w_0/v']) \mid s; v := w_0 \models \text{Skip})} \quad v' = \text{out}\alpha s \quad (\text{C.1.1})$$

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge (s; w_0 = e) \mid s; v := w_0 \models \text{Skip})} \quad (\text{C.1.2})$$

$$\frac{c \wedge (s; \text{pre } Op)}{(c \mid s \models Op) \xrightarrow{\varepsilon} (c \wedge (s; Op[w_0/v']) \mid s; v := w_0 \models \text{Skip})} \quad v' = \text{out}\alpha s \quad (\text{C.1.3})$$

$$\frac{c}{(c \mid s \models d!e \rightarrow A) \xrightarrow{d!w_0} (c \wedge (s; w_0 = e) \mid s \models A)} \quad (\text{C.1.4})$$

$$\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xrightarrow{d?w_0} (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)} \quad (\text{C.1.5})$$

$$\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models \text{var } x : T \bullet A) \xrightarrow{\varepsilon} (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)} \quad (\text{C.1.6})$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \text{let } x \bullet A_1) \xrightarrow{1} (c_2 \mid s_2 \models \text{let } x \bullet A_2)} \quad (\text{C.1.7})$$

$$\frac{c}{(c \mid s \models \text{let } x \bullet \text{Skip}) \xrightarrow{\varepsilon} (c \mid s; \text{end } x \models \text{Skip})} \quad (\text{C.1.8})$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1; B) \xrightarrow{1} (c_2 \mid s_2 \models A_2; B)} \quad (\text{C.1.9})$$

$$\frac{c}{(c \mid s \models \text{Skip}; A) \xrightarrow{\varepsilon} (c \mid s \models A)} \quad (\text{C.1.10})$$

$$\frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_1)} \quad \frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_2)} \quad (\text{C.1.11})$$

$$\frac{c \wedge (s; g)}{(c \mid s \models g \& A) \xrightarrow{\varepsilon} (c \wedge (s; g) \mid s \models A)} \quad (\text{C.1.12})$$

$$\frac{c}{(c \mid s \models A_1 \sqcup A_2) \xrightarrow{\varepsilon} (c \mid s \models (\text{loc } c \mid s \bullet A_1) \boxplus (\text{loc } c \mid s \bullet A_2))} \quad (\text{C.1.13})$$

$$\frac{c_1}{(c \mid s \models (\text{loc } c_1 \mid s_1 \bullet \text{Skip}) \boxplus (\text{loc } c_2 \mid s_2 \bullet A)) \xrightarrow{\varepsilon} (c_1 \mid s_1 \models \text{Skip})} \quad (\text{C.1.14})$$

$$\frac{c_2}{(c \mid s \models (\text{loc } c_1 \mid s_1 \bullet A) \boxplus (\text{loc } c_2 \mid s_2 \bullet \text{Skip})) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models \text{Skip})} \quad (\text{C.1.15})$$

$$\begin{array}{c}
(c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3) \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{loc } c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\text{loc } c_2 \mid s_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{\varepsilon} \left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{loc } c_3 \mid s_3 \bullet A_3) \\ \boxplus \\ (\text{loc } c_2 \mid s_2 \bullet A_2) \end{array} \right) \end{array} \right)
\end{array} \quad (C.1.16)$$

$$\begin{array}{c}
(c_2 \mid s_2 \models A_2) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3) \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{loc } c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\text{loc } c_2 \mid s_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{\varepsilon} \left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{loc } c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\text{loc } c_3 \mid s_3 \bullet A_3) \end{array} \right) \end{array} \right)
\end{array} \quad (C.1.17)$$

$$\begin{array}{c}
(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_3 \mid s_3 \models A_3) \quad 1 \neq \varepsilon \\
\hline
(c \mid s \models (\text{loc } c_1 \mid s_1 \bullet A_1) \boxplus (\text{loc } c_2 \mid s_2 \bullet A_2)) \xrightarrow{1} (c_3 \mid s_3 \models A_3)
\end{array} \quad (C.1.18)$$

$$\begin{array}{c}
(c_2 \mid s_2 \models A_2) \xrightarrow{1} (c_3 \mid s_3 \models A_3) \quad 1 \neq \varepsilon \\
\hline
(c \mid s \models (\text{loc } c_1 \mid s_1 \bullet A_1) \boxplus (\text{loc } c_2 \mid s_2 \bullet A_2)) \xrightarrow{1} (c_3 \mid s_3 \models A_3)
\end{array} \quad (C.1.19)$$

$$\begin{array}{c}
c \\
\hline
(c \mid s \models A_1 \llbracket x_1 \mid cs \mid x_2 \rrbracket A_2) \xrightarrow{\varepsilon} \left(\begin{array}{c} c \mid s \\ \models \\ (\text{par } s \mid x_1 \bullet A_1) \llbracket cs \rrbracket (\text{par } s \mid x_2 \bullet A_2) \end{array} \right)
\end{array} \quad (C.1.20)$$

$$\begin{array}{c}
c \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet \text{Skip}) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet \text{Skip}) \end{array} \right) \end{array} \right) \xrightarrow{\varepsilon} (c \mid (\exists x'_2 \bullet s_1) \wedge (\exists x'_1 \bullet s_2) \models \text{Skip})
\end{array} \quad (C.1.21)$$

$$\begin{array}{c}
(c \mid s_1 \models A_1) \xrightarrow{1} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee \text{chan } l \notin cs \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{1} \left(\begin{array}{c} c_3 \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_3 \mid x_1 \bullet A_3) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right)
\end{array} \quad (C.1.22)$$

$$\begin{array}{c}
(c \mid s_2 \models A_2) \xrightarrow{1} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee \text{chan } l \notin cs \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{1} \left(\begin{array}{c} c_3 \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\text{par } s_3 \mid x_2 \bullet A_3) \end{array} \right) \end{array} \right)
\end{array} \quad (C.1.23)$$

$$\begin{array}{c}
\left(\begin{array}{c} (c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \wedge (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \wedge (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \wedge (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \end{array} \right)
\end{array} \quad (C.1.24)$$

$$\begin{array}{c}
d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2 \\
\hline
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{d!w_2} \left(\begin{array}{c} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_3 \mid x_1 \bullet A_3) \\ \llbracket cs \rrbracket \\ (\text{par } s_4 \mid x_2 \bullet A_4) \end{array} \right) \end{array} \right)
\end{array}$$

$$\begin{array}{c}
(c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \\
\hline
d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2 \\
\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\text{par } s_2 \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right) \xrightarrow{d?w_2} \left(\begin{array}{c} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c} (\text{par } s_3 \mid x_1 \bullet A_3) \\ \llbracket cs \rrbracket \\ (\text{par } s_4 \mid x_2 \bullet A_4) \end{array} \right) \end{array} \right)
\end{array} \quad (C.1.25)$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2) \quad l \neq \varepsilon \quad \text{chan } l \notin cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{1} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (\text{C.1.26})$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{1} (c_2 \mid s_2 \models A_2) \quad l = \varepsilon \vee \text{chan } l \in cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (\text{C.1.27})$$

$$\frac{c}{(c \mid s \models \text{Skip} \setminus cs) \xrightarrow{\varepsilon} (c \mid s \models \text{Skip})} \quad (\text{C.1.28})$$

C.2 Specification traces operational semantics

The syntax of the labels of specification traces is given below, followed by the transition rules.

$$\begin{aligned}
 \textit{Label} &::= \textit{Pred} \mid \textit{Comm} \mid \textit{LAct} \\
 \textit{Comm} &::= \varepsilon \mid \textit{CName} \mid \textit{CName!Exp} \mid \textit{CName?VName} \\
 &\quad \mid \textit{CName?VName} : \textit{Pred} \\
 \textit{LAct} &::= \textit{VName}^* : [\textit{Pred}, \textit{Pred}] \mid \textit{Schema} \mid \textit{VName} := \textit{Exp} \\
 &\quad \mid \textit{var VName} : \textit{Exp} \mid \textit{var VName} := \textit{Exp} \mid \textit{end VName}
 \end{aligned}$$

$$\frac{(state(P_1) \models maction(P_1)) \xRightarrow{1} (state(P_2) \models maction(P_2))}{P_1 \xRightarrow{1} P_2} \quad (C.2.1)$$

$$\frac{c \wedge (s; p) \wedge (\exists v' \bullet s; Q)}{(c \mid s \models p \vdash Q) \xRightarrow{p \vdash Q}_P (c \wedge (s; Q[w_0/v']) \mid s; v := w_0 \models \text{Skip})} \quad v' = out\alpha s \quad (C.2.2)$$

$$\frac{c \wedge (s; \text{preOp})}{(c \mid s \models 0p) \xRightarrow{0p}_P (c \wedge (s; 0p[w_0/v']) \mid s; v := w_0 \models \text{Skip})} \quad v' = out\alpha s \quad (C.2.3)$$

$$\frac{c}{(c \mid s \models v := e) \xRightarrow{v := e}_P (c \wedge (s; w_0 = e) \mid s; v := w_0 \models \text{Skip})} \quad (C.2.4)$$

$$\frac{c \wedge (s; g)}{(c \mid s \models g \& A) \xRightarrow{g}_P (c \wedge (s; g) \mid s \models A)} \quad (C.2.5)$$

$$\frac{c}{(c \mid s \models d!e \rightarrow A) \xRightarrow{d!e}_P (c \wedge (s; w_0 = e) \mid s \models A)} \quad (C.2.6)$$

$$\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xRightarrow{d?x} (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)} \quad (\text{C.2.7})$$

$$\frac{c \wedge T \neq \emptyset \quad x \notin \alpha s}{(c \mid s \models \text{var } x : T \bullet A) \xRightarrow{(\text{var } x : T)}_P (c \wedge w_0 \in T \mid s; \text{var } x := w_0 \models \text{let } x \bullet A)} \quad (\text{C.2.8})$$

$$\frac{c}{(c \mid s \models \text{let } x \bullet \text{Skip}) \xRightarrow{(\text{end } x)}_P (c \mid s; \text{end } x \models \text{Skip})} \quad (\text{C.2.9})$$

$$\frac{(c_1 \mid s_1 \models A_1) \xRightarrow{1}_P (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \text{let } x \bullet A_1) \xRightarrow{1}_P (c_2 \mid s_2 \models \text{let } x \bullet A_2)} \quad (\text{C.2.10})$$

$$\frac{(c_1 \mid s_1 \models A_1) \xRightarrow{1}_P (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1; B) \xRightarrow{1}_P (c_2 \mid s_2 \models A_2; B)} \quad (\text{C.2.11})$$

$$\frac{c}{(c \mid s \models A_1 \llbracket x_1 \mid cs \mid x_2 \rrbracket A_2) \xRightarrow{\text{var } v_l, v_r := v, v}_P \left(\begin{array}{l} c \mid s; \text{var } v_l, v_r := v, v \\ \models \\ \left(\begin{array}{l} (\text{spar } v \mid v_l \mid v_r \mid x_1 := x_{l1} \bullet A_1[v_l/v]) \\ \llbracket cs \rrbracket \\ (\text{spar } v \mid v_r \mid v_l \mid x_2 := x_{r2} \bullet A_2[v_r/v]) \end{array} \right) \end{array} \right)} \quad \begin{array}{l} v' = \text{out} \alpha s \\ v = x_1, x_2 \\ \text{fresh } v_l, v_r \end{array} \quad (\text{C.2.12})$$

$$\begin{array}{c}
(c \mid s; \text{end } v, y \models A_1) \xRightarrow{1}_P (c_3 \mid s_3 \models A_3) \quad \text{chan } l = \varepsilon \vee \text{chan } l \notin cs \\
\hline
(c \mid s \models (\text{spar } v \mid x \mid y \mid x_1 := z_1 \bullet A_1) \llbracket cs \rrbracket (\text{spar } v \mid y \mid x \mid x_2 := z_2 \bullet A_2)) \\
\xRightarrow{1}_P \\
\left(\begin{array}{c} c_3 \mid s_3 \wedge s; \text{end } x \\ \models \\ \left(\begin{array}{c} (\text{spar } v \mid x \mid (\text{end } l), (\text{var } l) \mid y \mid x_1 := z_1 \bullet A_3) \\ \llbracket cs \rrbracket \\ (\text{spar } v \mid y \mid x \mid (\text{end } l), (\text{var } l) \mid x_2 := z_2 \bullet A_2) \end{array} \right) \end{array} \right)
\end{array} \quad (C.2.13)$$

$$\begin{array}{c}
(c \mid s; \text{end } v, y \models A_1) \xRightarrow{d?a}_P (c_3 \mid s_3 \models A_3) \\
(c \mid s; \text{end } v, x \models A_2) \xRightarrow{d!e}_P (c_4 \mid s_4 \models A_4) \\
\hline
d \in cs \quad c_3 \wedge c_4 \wedge \exists w_0 \bullet (s_3; (w_0 = a)) \Leftrightarrow (s_4; (w_0 = e)) \\
\hline
(c \mid s \models (\text{spar } v \mid x \mid y \mid x_1 := z_1 \bullet A_1) \llbracket cs \rrbracket (\text{spar } v \mid y \mid x \mid x_2 := z_2 \bullet A_2)) \\
\xRightarrow{d!e}_P \\
\left(\begin{array}{c} c_3 \wedge c_4 \wedge \exists w_0 \bullet (s_3; (w_0 = a)) \Leftrightarrow (s_4; (w_0 = e)) \mid s \\ \models \\ \left(\begin{array}{c} (\text{spar } v \mid x \mid a \mid y \mid x_1 := z_1 \bullet A_3) \\ \llbracket cs \mid \text{var } a := e \mid s_3 \wedge s_4 \wedge s; \text{end } x, y \rrbracket \\ (\text{spar } v \mid y \mid x \mid a \mid x_2 := z_1 \bullet A_4) \end{array} \right) \end{array} \right)
\end{array} \quad (C.2.14)$$

The above rule uses a new parallel construct that keeps track of the new input variable declared and the new state obtained as a consequence. It is used to ensure that, as required here, all transitions have a single label, and the label contains a guard, a communication, or an action.

The next rule ensures that in the next step of the evaluation of the parallelism, the variable declaration and state change are recorded. This concern was not present in (CAVALCANTI; GAUDEL, 2010b).

$$\begin{array}{c}
c \\
\hline
(c \mid s \models \left(\begin{array}{c} (\text{spar } v \mid x \mid y \mid x_1 := z_1 \bullet A_1) \\ \llbracket cs \mid \text{var } a := e \mid s_1 \rrbracket \\ (\text{spar } v \mid y \mid x \mid x_2 := z_2 \bullet A_2) \end{array} \right)) \\
\xRightarrow{\text{var } a := e}_P \\
(c \mid s_1 \models (\text{spar } v \mid x \mid y \mid x_1 := z_1 \bullet A_1) \llbracket cs \rrbracket (\text{spar } v \mid y \mid x \mid x_2 := z_2 \bullet A_2))
\end{array} \quad (C.2.15)$$

Rules similar to those above for parallelism are needed for external choice.

$$\frac{(c_1 \mid s_1 \models A_1) \xRightarrow{1}_P (c_2 \mid s_2 \models A_2) \quad \text{chan } 1 \notin cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xRightarrow{1}_P (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (\text{C.2.16})$$

Above, we assume that, if 1 is not a communication, then $\text{chan } 1$ is some special channel ε that does not belong to any synchronisation set cs .