# Incorporating Faults and Fault-Tolerance into Real-Time Networks: A graph-transformational approach

Daniel James Owen

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computing Science,
The University of Newcastle upon Tyne

June 2005

NEWCASTLE UNIVERSITY LIBRARY
204 26779 3
Thesis L8169

# Acknowledgements

I owe a debt of gratitude to my supervisor Dr. John Fitzgerald for shaping my ideas and for his continued help, guidance and encouragement throughout. Thanks are also due to Dr. Paul Ezhilchelvan who assumed my supervisor during John's encounter with the commercial world and Prof. Cliff Jones who has challenged every idea I have presented to him throughout, for without such discussions I would not have arrived at this point.

My thanks also extend to Dr. Stephen Paynter –my industrial supervisor– and Dr. Jim Armstrong in helping me understand the perils and pitfalls for formalising a real-time model.

I would also like to thank the Centre for Software Reliability for providing a convivial and stimulating research environment and the much needed coffee!

Thanks are due to BAE SYSTEMS for their sponsorship and intellectual support and for affording me the opportunity to evaluate my research in an industrial context. I am also grateful to the EPSRC for their financial support.

A special mention goes to my mother Barbara for her love and generosity and to my father for his encouragement throughout. I'd also like to challenge my Uncle Garth who has always wanted to read my thesis, to do so now and allow me the time to practice my golf! Thank-you all.

Finally to Karen, for the times we have had during the most demanding time of this project. You gave me perspective, encouragement and so much more. You tolerated my moods, and always offered an alternative outlook. And what's more, you did it with a smile!

# Contents

List of Tables . . . . . xi

List of Figures . . . . . xiii

**1 Introduction** **1**

1.1 Motivation . . . . . 2

1.2 Contribution . . . . . 2

1.2.1 Structure . . . . . 2

1.2.2 Design Transformation . . . . . 4

1.2.2.1 Suitability of Graph Grammars as transformational method . . . . . 4

1.2.3 Faults in RTNs . . . . . 5

1.2.4 Soundness . . . . . 5

1.2.4.1 Axiomatic Semantics . . . . . 5

1.2.4.2 Operational . . . . . 6

1.2.5 Summary . . . . . 6

1.3 Thesis Argument . . . . . 6

1.4 Thesis Structure . . . . . 7

## I Preliminaries 9

**2 Background - Faults, Fault Tolerance and Fault Treatment** **10**

2.1 Faults . . . . . 11

2.1.1 System Structure . . . . . 11

2.1.2 Specification and Design . . . . . 13

2.1.3 Fault Classification . . . . . 17

2.2 Fault Hypothesis . . . . . 19

2.3 Fault-Tolerance . . . . . 20
2.3.1 Principles of Fault Tolerance . . . . . 20
2.3.2 Formalisms of Fault Tolerance . . . . . 22
2.4 Real-Time Networks . . . . . 23
2.4.1 RTN-SL . . . . . 24
2.4.2 Verification of Real-Time Networks . . . . . 28
2.5 Methodology . . . . . 29
2.5.1 The SHARD Analysis Method . . . . . 29
2.5.1.1 Method . . . . . 29
2.5.1.2 Guide Words . . . . . 30
2.5.2 Design Transformations . . . . . 34
2.6 Conclusions . . . . . 34

**3 RTN Graph Grammar Transformations** **36**
3.1 Introduction . . . . . 36
3.2 The Grammar for RTN-SL . . . . . 39
3.2.1 RTN Production Rules . . . . . 39
3.3 Design Transformations using Graph Grammars . . . . . 40
3.4 An Example derivation using the graph grammar . . . . . 43
3.4.1 Scenario . . . . . 43
3.4.2 Specification . . . . . 43
3.4.3 Design . . . . . 44
3.4.3.1 RTN-SL Design Fragment . . . . . 44
3.4.4 Faults . . . . . 45
3.4.5 The chosen Template . . . . . 47
3.4.6 Transformation . . . . . 49
3.5 Evaluation . . . . . 50

**II Semantic Descriptions** **52**

**4 Fault Semantics** **53**
4.1 Real Time Logic . . . . . 53

4.1.1 RTL Event Model . . . . . 53

4.1.2 Syntax . . . . . 54

4.2 Faults in Real-Time Networks . . . . . 56

4.2.1 Fault Hypotheses . . . . . 57

4.2.1.1 State Machine Component Faults . . . . . 60

4.2.1.2 Activity Component Faults . . . . . 65

4.2.1.3 IDA Component Faults . . . . . 65

4.3 Extensions to the RTN-SL language . . . . . 65

4.3.1 Language Extensions . . . . . 66

4.3.1.1 Multiple Static State Exit Transitions . . . . . 66

4.3.1.2 Fault Transitions . . . . . 66

4.3.1.3 *Speculative* Reads . . . . . 67

4.3.2 Extended (Axiomatic) Semantics ($\Omega_f$) . . . . . 70

4.3.2.1 State-Machines, $\Omega_{SM_f}$ . . . . . 71

4.3.2.2 Operations, $\Omega_{Op_f}$ . . . . . 75

**5 Structural Operational Semantics** 76

5.1 Basics . . . . . 77

5.1.1 Labelled Transition Systems . . . . . 77

5.1.1.1 Term Algebras . . . . . 77

5.1.1.2 Transition System Specifications . . . . . 78

5.1.2 Plotkin-style Transition Rules . . . . . 78

5.2 An Operational Model for RTNs . . . . . 78

5.2.1 RTN-SL SOS Rules . . . . . 81

5.2.1.1 A TSS Specification for RTNs, $T_0$ . . . . . 82

5.2.1.2 Fault Transition System Specification for RTNs, $T_1$ . . . . . 87

5.3 An Operational Conservative Extension . . . . . 89

5.3.1 Definitions . . . . . 90

5.3.2 Faults are a conservative extension . . . . . 91

5.3.3 Animating SOS rules with LETOS . . . . . 92

**6 A soundness argument for the axiomatic semantics** 93

6.1 Objectives . . . . . 93

6.2 Approach . . . . . 94

6.2.1 An Example . . . . . 94

6.2.1.1 Axiomatic Specification . . . . . 95

6.2.1.2 Operational Model . . . . . 95

6.2.2 Correctness Specification . . . . . 96

6.2.3 An informal argument . . . . . 98

6.3 Trace Induction . . . . . 100

6.3.1 Model Space . . . . . 101

6.3.2 Induction Rules . . . . . 102

6.4 RTN Soundness argument . . . . . 105

6.4.1 Ax6 . . . . . 105

6.5 Review . . . . . 110

**III Towards Tolerance** **111**

**7 Showing enhancements via graph grammars** 112

7.1 Passive State Replication Template . . . . . 114

7.1.1 Description . . . . . 114

7.1.2 Transformation . . . . . 114

7.1.3 Design . . . . . 118

7.1.4 Rigorous Proof . . . . . 119

7.2 Triple Modular Redundant Template . . . . . 121

7.2.1 Transformation . . . . . 121

7.2.2 Design . . . . . 122

7.2.3 Rigorous Proof . . . . . 124

7.2.4 Permutations . . . . . 125

7.3 Watchdog Timer Template . . . . . 127

7.4 Template Variations . . . . . 129

7.4.1 Temporal Redundant IDA . . . . . 129

7.4.2 Fail-Signal (Activity) . . . . . 130

7.4.3 Fail Stop (Activity) . . . . . 131

**8 Case Study - "A BVRAAM Launch System"** **132**

8.1 Identification . . . . . . . . . 133

8.2 Design overview . . . . . . . . . 133

8.2.1 Abstract Design description . . . . . . . . . 133

8.3 The Initial (Host) Design . . . . . . . . . 136

8.3.1 Transformations . . . . . . . . . 137

8.3.1.1 Refining the *Read Aircraft Messages*, ($RAM_a$) Activity . . . . . . . . . 137

8.3.1.2 Refining the *Transfer Alignment*, ($TA_a$) Activity . . . . . . . . . 138

8.3.2 RTN-SL Specification . . . . . . . . . 145

8.4 A SHARD Analysis . . . . . . . . . 145

8.4.1 Guide Words . . . . . . . . . 145

8.4.2 Analysis Results . . . . . . . . . 145

8.5 Fault Hypotheses for the BVRAAM Launcher Design . . . . . . . . . 150

8.5.1 RTL Specifications of $FH_i$ . . . . . . . . . 150

8.5.2 RTL Safety Theorems for $FH_i$ . . . . . . . . . 152

8.6 The Transformed Design . . . . . . . . . 152

8.6.1 Transformations . . . . . . . . . 152

8.6.1.1 Addressing $FH_1$ . . . . . . . . . 152

8.6.1.2 Addressing $FH_2$ and $FH_5$ . . . . . . . . . 154

8.6.1.3 Addressing $FH_3$ . . . . . . . . . 154

8.6.1.4 Addressing $FH_4$ . . . . . . . . . 154

8.7 Axiomatic Semantics generated by $\Omega_f$ . . . . . . . . . 154

8.8 Validation . . . . . . . . . 154

8.9 Evaluation . . . . . . . . . 160

**IV Evaluation** **161**

**9 Conclusions & Further Work** **162**

9.1 Summary . . . . . . . . . 162

9.2 Evaluation of RTN-SL for specifying faults . . . . . . . . . 163

9.2.1 RTN-SL Language Extensions . . . . . . . . . 164

9.2.2 Fault Semantics, $\Omega_f$ . . . . . . . . . 164

9.3 RTN Operational Semantics . . . . . 164
9.3.1 SOS & LTS . . . . . 164
9.3.2 Conservative Extension . . . . . 165
9.3.3 Soundness . . . . . 165
9.4 Suitability of graph grammars and application of methodology . . . . . 166
9.4.1 Transformational methodology . . . . . 166
9.4.2 Case Study Evaluation . . . . . 166
9.5 Further Work . . . . . 167
9.6 Epilogue . . . . . 168

**Bibliography** 169

**V Supplementary Material** 176

**A Case Study Specifications** 177
A.1 RTN-SL Design . . . . . 178
A.2 Modified Axiomatic Semantics . . . . . 186

**B Axiomatic Soundness Proofs and Lemmas** 189
B.1 Soundness Conjectures . . . . . 189
B.1.1 Ax2 :: Static State Exit . . . . . 190
B.1.2 Ax3 :: Dynamic State Exit . . . . . 190
B.1.3 Ax4 :: (Initial) State Entry . . . . . 190
B.1.4 Ax5 :: State Entry . . . . . 190
B.1.5 Ax6 :: (Dynamic) State Progress . . . . . 190
B.1.6 Ax7 :: (Static) State Progress, Event Transition . . . . . 191
B.1.7 Ax8 :: (Static) State Progress, Timed Transition . . . . . 191
B.1.8 Ax9 :: (Dynamic) State Stability . . . . . 191
B.1.9 Ax10:: (Static) State Stability, Event Transition . . . . . 191
B.1.10 Ax11:: (Static) State Stability, Timed Transition . . . . . 191
B.1.11 Ax12:: Stop States . . . . . 191
B.2 Top Level Soundness Proofs . . . . . 191
B.2.1 Ax2 . . . . . 191

B.2.2 Ax3 . . . 192
B.2.3 Ax4 . . . 192
B.2.4 Ax5 . . . 193
B.2.5 Ax6 . . . 194
B.2.6 Ax8 . . . 194
B.2.7 Ax9 . . . 194
B.2.8 Ax10 . . . 195
B.3 Additional Lemmas . . . 195

**C VDM Tool support for SOS definitions** **197**
C.1 Abstract Syntax . . . 197
C.2 Context Conditions . . . 199
C.2.1 Auxiliary Objects . . . 199
C.3 (Normative) Semantics . . . 199
C.3.1 Semantic Objects . . . 199
C.3.2 Auxiliary Functions . . . 200
C.3.3 Semantic Rules . . . 200
C.3.3.1 Auxiliary Semantic Functions . . . 201
C.3.3.2 Operations . . . 202
C.3.3.3 States . . . 207
C.3.3.4 Transitions . . . 212
C.3.3.5 Ports . . . 215
C.3.3.6 IDAs . . . 216
C.4 Fault Semantics . . . 220
C.4.1 Late Exit Fault . . . 221
C.4.1.1 Static State . . . 221
C.4.1.2 Dynamic State . . . 223
C.4.2 Early Exit Fault . . . 224
C.4.2.1 Static State . . . 224
C.4.2.2 Dynamic State . . . 226
C.4.3 Read Faults . . . 227
C.4.4 Write Faults . . . 232

C.4.5 Crash Faults . . . . . 237

C.5 Implementation Considerations . . . . . 237

C.5.1 An Executable Specification (for Animation) . . . . . 238

C.5.1.1 Modelling non-determinism . . . . . 238

C.5.1.2 Implementation in the VDMTools . . . . . 239

# List of Tables

2.1 The Basic Protocols . . . . . . . . . . 25

2.2 Fault classes and detectability . . . . . . . . . . 31

2.3 Table of guide words applicable to RTNs . . . . . . . . . . 32

2.4 Fragment of an SHARD Analysis . . . . . . . . . . 33

4.1 Bounds to Fault intervals . . . . . . . . . . 57

4.2 Faults considered . . . . . . . . . . 59

7.1 Faults previously considered in Chapter 2 . . . . . . . . . . 113

8.1 Mode Encoding . . . . . . . . . . 133

8.2 Previous/Next Mode Table . . . . . . . . . . 134

8.3 Table of guide words applicable to RTNs . . . . . . . . . . 145

8.4 SHARD analysis of flow $AIM_i$ of the BVRAAM Launch system top level design . . . . . 147

8.5 SHARD analysis of flow $MS2_i$ of the BVRAAM Launch system top level design . . . . . 148

8.6 SHARD analysis of flow $MS1_i$ of the BVRAAM Launch system top level design . . . . . 149

8.7 SHARD analysis of flow $MB_i$ of the BVRAAM Launch system top level design . . . . . . 149

# List of Figures


1.1 First level overview . . . . . 4

1.2 Overiew of each goal . . . . . 6

2.1 Duplex communication medium . . . . . 13

2.2 Erroneous Transitions and Erroneous States . . . . . 15

2.3 Specification and Implementation Model . . . . . 17

2.4 Omission Fault *Observable* Intervals . . . . . 18

2.5 Observable fault intervals . . . . . 19

2.6 Weakening the specification . . . . . 20

2.7 Passive State Replication . . . . . 22

2.8 Dynamic State . . . . . 26

2.9 Composite Dynamic State . . . . . 26

2.10 An Example Graphical RTN-SL Network . . . . . 27

2.11 Existing Semantic Framework . . . . . 28

2.12 A SHARD invoked design process . . . . . 30

3.1 Production rule, $p$ . . . . . 37

3.2 Host graph, $H$ . . . . . 38

3.3 Result Graph, $\overleftarrow{H}$ . . . . . 38

3.4 The RTN-SL Initial Graph . . . . . 39

3.5 The A_IDA *family* based production rules and embedding relations . . . . . 40

3.6 Context Sensitive SM productions and embedding relations . . . . . 41

3.7 Passive state template . . . . . 42

3.8 Abstract System Specification . . . . . 43

3.9 A Target Tracker . . . . . 44

3.10 Host Graph, $G_H$ . . . 46
3.11 Production Rule Context . . . 47
3.12 Graph Grammar Syntax of PSR Template . . . 48
3.13 Result graph net1' . . . 49
3.14 Required Semantic Framework . . . 50

4.1 Repeat of Figure 2.5: Observable fault intervals . . . 56
4.2 A Dynamic State . . . 58

5.1 Possible derivation tree from some starting configuration, $(\sigma_1, \pi_1)$ . . . 80
5.2 Possible sequencing of actions . . . 80

6.1 Example fragment of an RTN-SL Design . . . 95

7.1 Template #1 - Passive state replication . . . 115
7.2 Template #2 - Triple Modular Redundancy . . . 121
7.3 Template #3 - A Watchdog Timer . . . 127
7.4 Template #4 - Temporal Redundant IDA . . . 130
7.5 Template #5 - Fail Signal . . . 131

8.1 A BVRAAM Launcher Design . . . 134
8.2 Host Graph . . . 136
8.3 Neighbourhood of $RAM_a$ . . . 137
8.4 $RAM_a$ - p1 . . . 138
8.5 $RAM_a$ - p2 . . . 139
8.6 $RAM_a$ - p3 . . . 139
8.7 $RAM_a$ - p4 . . . 140
8.8 $RAM_a$ - p5 . . . 141
8.9 $RAM_a$ - p6 . . . 142
8.10 Neighbourhood of $TA_a$ . . . 142
8.11 $TA_a$ - p1 . . . 143
8.12 $TA_a$ - p2 . . . 143
8.13 $TA_a$ - p3 . . . 144
8.14 $TA_a$ - p4 . . . 144
8.15 $TA_a$ - pFT1 . . . 153

# Author's Declaration

All work contained within this thesis represents the original contribution of the author. However, some material presented has previously appeared in the following:

Dan J Owen and Paul D Ezhilchelvan, *Verifiable fault-tolerant transformation of a real-time legacy system*, Tech. Report CS-TR-785, University of Newcastle, 2002

# Chapter 1

# Introduction

## Contents


1.1 **Motivation** . . . . . . . . . . 2
1.2 **Contribution** . . . . . . . . . . 3
1.2.1 Structure . . . . . . . . . . 3
1.2.2 Design Transformation . . . . . . . . . . 5
1.2.3 Faults in RTNs . . . . . . . . . . 6
1.2.4 Soundness . . . . . . . . . . 6
1.2.5 Summary . . . . . . . . . . 7
1.3 **Thesis Argument** . . . . . . . . . . 8
1.4 **Thesis Structure** . . . . . . . . . . 9


The introduction of fault tolerance into real-time systems presents particular challenges because of the price of redundancy and the added complexity of verification and validation on these redundant structures. This thesis brings structural and formal design techniques to bear on this problem.

Verification of fault tolerance properties in such systems has only received limited attention, in particular the design methodologies are in their infancy. We propose a transformational design methodology, specific to a real-time systems architecture. We then reason about the compositional addition of fault tolerant components and templates of the derived designs. This requires that we show the existing axiomatic semantics for our chosen architecture sound with respect to a more constructive semantic model. The issues of presenting an operational model for a real-time architecture are discussed and a model is proposed. The extension of the existing semantics, to allow for faulty behaviour, is shown to preserve the existing semantic properties and the application of our methodology shown to be usable by a sizeable study.

The contribution of this thesis is to define a transformational design methodology in which components can be extracted from a design and replaced by another component preserving functionality while providing fault tolerance. This approach requires the precise modelling of the faults we consider, the transformational method and verification of the transformed design with respect to faults.

## 1.1 Motivation

Despite the wide spread awareness of the importance of fault tolerance and dependability, many industrial-strength, real-time systems have stopped short of exploiting even well-known fault tolerance techniques. Missile systems fall into this category of systems in so far as the UK defence industries are concerned. The primary reason for this situation is the consequence of the constraints that are normally placed on missile systems. These constraints are two-fold.

First comes the space constraint which discourages the deployment of redundant processors or power supply; any extra space is likely to be used up by processors with software that can provide highly sophisticated functions. Non-redundant power supply also mitigates against processor replication since a chain can only be as strong as its weakest link. Secondly, there is the timeliness constraint. Missile systems are basically control systems. This means that there are hard deadlines within which the targets have to be identified and their co-ordinates correctly estimated based on the radar data.

Despite these constraints, efforts are being made to incorporate fault tolerance in missile systems due mainly to the growing importance placed on the UK Ministry of Defence regulations [Min99, Min97, Min96] on ordnance safety and reliability. Decreasing size and cost of hardware components and the increasing power and speed of such devices also motivate such efforts. The work reported here is a part of the on-going work being carried out to make the legacy missile systems of MBDA UK more fault tolerant without violating the timeliness constraints.

The work is simultaneously motivated towards future system design, where it is feared people may shy away from applying fault tolerant strategies to real-time systems because of the perceived complexity and the lack of a sound engineering basis to help master the complexity. This work aims to provide such a sound engineering basis for fault tolerance strategies in a particular class of Real-Time Networks, and explore its utility.

## 1.2 Contribution

In this section we first describe the ultimate aim of our research and work backwards from this to determine the subgoals that must be achieved. We thereby introduce the main threads of the work in the thesis and identify the specific contributions in each thread.

### 1.2.1 Structure

Our goal is to provide a framework supporting (i) the transformation of existing real-time system designs to introduce tolerance of specified faults and (ii) the argument that the transformed design satisfies its original specification, extended with the specified faults. In order to achieve this goal, we must provide (i) a transformation design methodology based upon some design notation; (ii) a design notation that is capable of specifying fault tolerant techniques; (iii) a sound semantic model to reason upon and (iv) a specification of the faults we wish to tolerate.

The specific real-time system architecture that is used in this study is that of Real Time Networks (RTNs). A real-time network is a system of concurrent processes that can only communicate via defined, explicit

paths. RTNs are described using the Real Time Network Specification Language (RTN-SL), a formal specification and design language covering the behaviour of RTNs in both the value and time domains. RTN-SL has been designed to be integrated into the MASCOT-3 design notation [IoMJ87] which is used within MBDA. The RTN-SL is compatible with the process interaction protocols developed by MBDA [Sim03] and subsumes the Activity Description Language (ADL) [PAH00], the formal specification language developed for describing the behaviour of the individual activities in RTNs. RTN-SL has a formal definition of syntax given in VDM-SL [LHP+96] and a formal axiomatic semantics given by means of a semantic function $\Omega$ taking RTN-SL designs to sets of axioms expressed in Real-time logic (RTL). This allows us to verify designs against specifications expressed in RTL. In order to support our goal of fault-tolerant design in RTN-SL, we need to extend the languages syntax and semantics with features to facilitate the specification of fault-tolerant techniques.

Given that there exists a specification language for RTNs for which it is possible to reason that some design satisfies some specification, we must consider the implications of adding the ability to describe faulty behaviours to the design methodology. Suppose we have a RTN-SL design $D$ that purportedly satisfies some specification spec. We write this as

$D$ *sat spec*

We identify potential faulty behaviours and wish to define a transformation from $D$ to a fault tolerant version $D_{FT}$ that satisfies some weaker specification $spec_f$ that allows for the behaviours of faults. We wish to show

$D_{FT}$ *sat* $spec_f$

In order to achieve this we require:

- A means of describing the transformation $D$ to $D_{FT}$. The work required to achieve this is described further in Section 1.2.2
- A means of specifying the faulty behaviours to be tolerated in $spec_f$. This is described further in Section 1.2.3
- A means of showing the correctness of the design transformation with respect to the extended specification $spec_f$, for which we require a sound semantic basis for RTN-SL taking account of faulty components. The work required to achieve this described further in Section 1.2.4. We term this semantic basis $\Omega_f$.

Figure 1.1 illustrates this aim and its immediate subgoals. Each of the leaf nodes are described further below and outline either the technologies we utilize or the contributions of work we make.

An underlying principle of our work will be the use of formal methods. This is intende to bring significant benefits to the work. First, it will strengthen the argument that there is a sound basis to the engineering approach advocated. Second, and more importantly, it will support rigorous, repeatable analysis. Formalisation will help to reduce ambiguity and provide a basis from which to present and discuss our work. The requirement for a formal argument is called for in the MoD Standard, 00-55 [Min97] which has informed the development of many of the systems to which we hope our work will be applicable. The standard requires of a formal method:

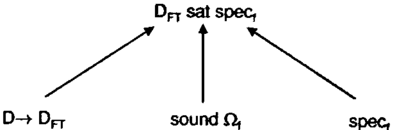



Figure 1.1: First level overview

> *A software specification and production method, based on mathematics, that comprises: a collection of mathematical notations addressing the specification, design and development processes of software production; a well-founded logical inference system in which formal verification proofs and proofs of other properties can be formulated; and a methodological framework within which software may be developed from the specification in a formally veritable manner.* [Min97]

Each of the three subgoals identified so far has a strand of work reported in the thesis. In the following sections, we describe the work to be done in each strand.

### 1.2.2 Design Transformation

Numerous design methods exist that enable the stepwise development of real-time systems [BSS94, LJ92]. MASCOT-3, which uses RTN-SL as its specification language, is one such. At a network level, one can first identify the concurrent components and the communication paths between them. Furthermore, the nature of the communication can be identified, and this specifies the nature of the asynchrony or synchrony between components. From a RTN-SL specification, the designer is free to choose a design or implementation structure. Common design patterns are known [Bor98] and are left to a designer to apply correctly and at will in any scenario. No methodology exists either for modifying existing designs to apply well known solutions to common issues, such as jitter control in a real-time system, or in applying fault tolerant solutions. We therefore require a methodological solution to allow for a transformational method to apply pre-defined design templates, for both common design patterns and fault tolerance strategies.

#### 1.2.2.1 Suitability of Graph Grammars as transformational method

We propose to use graph grammars to define design templates, particularly of fault tolerance strategies, to support the transformation of RTN-SL designs to more tolerant designs for a specified fault hypothesis. A RTN-SL design would be represented by a graph whose nodes represent the components of the RTN and whose links represent the control or data flows between components. The transformations defined by production rules in the graph grammar remove a node, or group of nodes, from a host design and replaces it with a new arrangement. The existing connections from the host design to the removed components are replaced with new ones, as specified by the production rule. Each production rule may have an associated context restriction, limiting its application to only those situations to which the transformation is deemed appropriate.

The goal of our work in this strand is to use graph grammars to define a compositional methodology which restrains the effect on the design to those components, or sub-systems, being transformed and should assist in the verification of the transformed design.

### 1.2.3 Faults in RTNs

Modelling and reasoning about faults has not yet been considered in the development of RTN-SL. There are three principal reasons for this. First, the past experience of real-time system designers within the UK Defence Industry emphasises correctness by construction over fault tolerance. The past trend to use bespoke components further supports this. However trends towards the use of off-the-shelf (OTS) components is tending to change this, and the need to tolerate component failures that may be outside a designers control must now be considered. As a result, faults have rerely been considered and are therefore an unknown entity. We have noted approaches in other real-time systems, notably those of a time-triggered architecture (TTA) [SHS$^+$97], but the RTN-SL approach to modelling events has not to date considered erroneous events. RTN-SL is still in its infancy and does not yet have a mature design methodology. Considering faults is yet another extension, which is expected to necessitate changes to the RTN-SL language semantics.

The goal of our work in this strand is to provide a way of specifying faulty behaviours which can be combined with specifications of normal behaviours so that the combination can be shown to be satisfied by some more tolerant design.

### 1.2.4 Soundness

Central to our thesis is a solid semantic framework. Although there already exists an axiomatic semantic for RTNs, $\Omega$ [Pay02], it remains to be shown sound. We first extend the axiomatic semantics to accommodate the language extensions necessary to permit the design of fault-tolerance mechanisms and attempt to show that this extension to the existing axiomatic semantics is a conservative one which preserves the properties of the original terms. We then show soundness of the axiomatic semantics, using an operational semantic model.

#### 1.2.4.1 Axiomatic Semantics

The axiomatic semantics gives the meaning of a model (specification or design) by giving the properties that characterise the meaning. For example, the axiomatic semantics for RTNs states that, if a read occurs on an IDA, then a write must have occurred to the same IDA previously. It is known that the main outstanding work on the RTN-SL axiomatic semantics is to prove it sound with respect a more constructive semantics. This would contribute to the confidence in the consistency of the axioms, and help convince the wider community of the soundness of the RTN-SL approach, and its suitability for use on safety-critical and mission-critical systems. However, it has been unclear how to define an operational or denotational semantics for RTNs [Pay02].

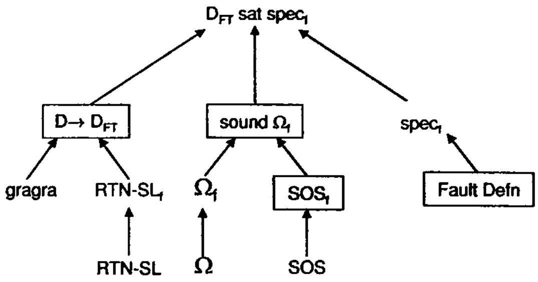



Figure 1.2: Overiew of each goal

#### 1.2.4.2 Operational

An operational semantics is a set of rules which provide a more constructive framework, specifying how the state of an actual or hypothetical computer changes while executing a program. We choose a specific form of operational semantics called structural operational semantics (SOS) [Plo81, Plo03] because of its intuitive appeal and flexibility, and because SOS has found considerable application in the study of the semantics of concurrent processes. A SOS characterises the meaning of a program by defining a labelled transition system, whose transitions are between program or system states. We later show the additional axioms to the RTN-SL language are new transitions.

Our goals for this strand of work are first to specify an operational semantic model for RTNs and then to derive proof tactics in which to show an axiomatic semantics is sound with respect to the operational semantics.

### 1.2.5 Summary

From the overall aims of this thesis and the sub-goals identified above, it should be clear that several contributions will be made to complete the thesis. Each of these contributions are separated into individual threads of work and brought together to achieve our aim: to reason that a transformed design does still satisfy the original specification given some set of fault definitions.

Figure 1.2 illustrates the underlying technologies used for each thread or work, or further sub-goals that must, in turn, be achieved.

## 1.3 Thesis Argument

We assert that classical fault tolerance strategies can be incorporated into RTNs using a transformational approach in such a way that the the transformations preserve the required functionality and provide tolerance to a specified set of faults. The aim is to provide a semantic framework to support reasoning about these transformations.

Practitioners use a range of design methodologies to model critical real-time systems. For many applications, these methods must have a well-defined syntax and semantics. Practitioners must also have the

ability to conduct safety assessments over these models, such as fault tree and failure mode analyses. Most of the above have tool support, however do not have a framework to relate the results of one into the other, for example, identifying a critical fault then suggesting a design pattern to tolerate it.

We argue that transformations, expressed over a graph grammar syntax, do integrate hypothetical fault analysis and design techniques. Further, providing a semantic framework allows to reason about such emergent properties, such as fault tolerance. Such transformations are constrained to respect the well-formedness of designs and their design language principles.

We also argue that extending the existing semantics to make faults explicit, does not violate existing properties of a design.

To substantiate our argument, we do the following:

- Specify first the abstract definition of faults, then the definition of each plausible fault with respect to each RTN components;
- Develop a (context-sensitive) graph grammar to facilitate the transformations we propose;
- Construct an operational semantics for RTNs and argue that the existing axiomatic semantics is sound with respect to the operational model, therefore validating the assertion all results shown previously are valid;
- Demonstrate the methodology of transforming an existing RTN-SL design to a more fault tolerant one given a set of hypothetical faults using material supplied by practitioners, showing the formal arguments for the fault tolerance are sound.

## 1.4 Thesis Structure

The thesis is structured into four parts: Premilinaries, Semantic Descriptions, Towards Tolerance and Evaluation. These four parts fit with the described overview and reflect the approach taken in this thesis. Part II explores the existing semantic descriptions for RTNs, faults and fault tolerance, whilst Part III illustrates through a case study how design transformations can be achieved using a graph grammar. Part IV evaluates the work and asses whether we have achieved our overal goal.

Part I includes **Chapter Two**, where we provide an overview of the existing literature definitions of faults and fault tolerance. For the readers orientations, we outline the RTN-SL specification language for RTNs: its features and graphical syntax. A hypothetical safety analysis technique is reviewed, for examining the effects of each fault class at each components interface, a process which forms the 'fault hypothesis' for a design. **Chapter Three** briefly introduces a graph grammar approach to design transformations, then outlines a 24-rule grammar for RTNs. This grammar satisfies two requirements: First, all well-formed RTNs are reachable from the grammar's initial graph, secondly, classical fault tolerant strategies can be applied to existing or on-going designs in a compositional manner.

Part II contains **Chapters Four** and **Five** which define the semantic framework required to support the rigorous arguments presented as our case study in Chapter Eight. Chapter Four specifies the definition of faults we consider plausible for RTNs, then propose an extension to the existing axiomatic semantics

for RTNs. Chapter Five proposes the first attempt at giving RTNs an operational semantics which is the used in **Chapter Six** to argue the existing, then the extended, axiomatic semantics sound with respect the operational model.

The fault tolerant templates we propose are documented in **Chapter Seven**, which begins Part III, and their applicability demonstrated in **Chapter Eight**, which features a case study that illustrates the applicability of our methodology by applying the fault tolerant templates from Chapter Seven to tolerate a subset of the faults identified in Chapter Four.

We conclude the thesis in Part IV with**Chapter Nine** which presents conclusions drawn from the thesis and the extent to which work in the previous chapters supports our thesis argument. A number of areas for possible future work are also highlighted.

**Appendix A** presents the complete PVS specification of our case study example, both the PVS model and the theorems and their proofs which demonstrate the transformed design is tolerant to the faults considered. **Appendix B** presents the complete soundness argument for the axiomatic semantics with respect the operational model.

**Appendix C** is a complete presentation of the SOS rules for RTN-SL, accompanied by their embedding in a VDM-SL model to syntax and type check each rule and an investigation into the implementation issues of the concurrent, non-deterministic operational semantics. **Appendix D** concludes with the entire presentation of the RTN-SL graph grammar.

# Part I

# Preliminaries

# Chapter 2

# Background - Faults, Fault Tolerance and Fault Treatment

## Contents


**2.1 Faults** . . . . . . . . . . 14
- 2.1.1 System Structure . . . . . . . . . . 14
- 2.1.2 Specification and Design . . . . . . . . . . 17
- 2.1.3 Fault Classification . . . . . . . . . . 22

**2.2 Fault Hypothesis** . . . . . . . . . . 26

**2.3 Fault-Tolerance** . . . . . . . . . . 27
- 2.3.1 Principles of Fault Tolerance . . . . . . . . . . 27
- 2.3.2 Formalisms of Fault Tolerance . . . . . . . . . . 30

**2.4 Real-Time Networks** . . . . . . . . . . 32
- 2.4.1 RTN-SL . . . . . . . . . . 33
- 2.4.2 Verification of Real-Time Networks . . . . . . . . . . 37

**2.5 Methodology** . . . . . . . . . . 38
- 2.5.1 The SHARD Analysis Method . . . . . . . . . . 38
- 2.5.2 Design Transformations . . . . . . . . . . 44

**2.6 Conclusions** . . . . . . . . . . 45


In this chapter, we outline the existing work on faults, fault tolerance and fault-treatment which is applicable to real-time systems and to a design transformation methodology. Section 2.1 details and classifies the fault specifications we consider. Section 2.3 describes classical principles of fault tolerance with which to correct a system exhibiting faults. Existing approaches are surveyed in Section 2.3.2 where we examine how faults, and more importantly fault tolerance, is formalised. We look at how one reasons that first, a fault tolerant system design is functionally equivalent to its non-fault tolerant counterpart, and secondly, that it has the fault tolerant characteristics (or behaviour) it claims. Section 2.4 looks at our target design method, Real-Time Networks, for which we begin the first investigations of faults of the design method.

Section 2.2 suggests a hypothetical design analysis technique to draw a *fault hypothesis* from a system design. The results of this analysis can then either prompt a re-design or suggest the use of some standard fault tolerance strategies, presented as design templates, which can be applied as transformations to the analysed design. A methodology for such transformations is suggested in Section 2.5.2

We conclude this chapter by highlighting the scope and motivation for the work described in subsequent chapters.

## 2.1 Faults

This section does not deal with faults in RTNs *per se* but is aimed at giving informal but precise definitions of faults in computing systems. It is a summary of the work undertaken within the "Reliable and Fault Tolerant Computing" scientific and technical community [ES85, Cri85, LA90, Lap92] which strives to propose clear and widely acceptable definitions for basic fault tolerance concepts.

### 2.1.1 System Structure

The advantage of developing a simple system model is to provide a basic framework with respect to which various aspects of structure, design and fault tolerance can be examined. More importantly, this examination can be independent of whether hardware or software systems, or sub-components of either, are being considered. We proceed by describing our simple system model.

First, the notion of a system should be given a more precise meaning - a *system* is an identifiable mechanism which maintains a pattern of behaviour at an interface between the mechanism and its environment. An *interface* exists at the system boundary at which interaction between two systems occurs. A system is said to interact with its *environment* and responds to stimuli at the interface between the system and the environment. It is not the system which constrains the interaction but the interface [dH01]. Consider, for example, a division system with inputs $x$ and $y$ and an output $z$. Whilst a system description may be

$$x \in \mathbb{R} \wedge y \in \mathbb{R} \setminus \{0\} \;\Rightarrow\; z = x/y$$

which asserts that "*if* the environment provides proper inputs, *then* the system produces the desired result", an interface description may be

$$x \in \mathbb{R} \wedge y \in \mathbb{R} \setminus \{0\} \wedge z = x/y$$

which asserts that "the environment provides proper inputs and the system produces the desired result".

Requiring the interface, between an environment or system and components, to be constant provides for a compositional design methodology. From this, we can reason about each component in a *black-box* approach to derive system properties which allows that a component may be replaced by alternative ones which provide the same service at their common interfaces. Such a requirement is a common assumption of many object-oriented design approaches.

The environment of a system is considered as yet another system which provides input to and receives output from the first system; thus the system can provide a service in response to requests from the environment. The external behaviour of a system can be described in terms of a finite set of states, the external

states of the system. At discrete instants of time the system makes a transition from one external state to another, and thus moves through a sequence of external states which generate externally observable events. Thus, a system maintains behaviour at the interface between the system and its environment.

Systems implement their service (which exhibits the behaviour required by the specification) by using the services of other systems and components. A system, $u$ depends on a system/component, $r$ if the correctness of $u$'s behaviour depends on the correctness of $r$'s behaviour. If a system depends on lower-level systems/components to correctly provide its service, then a failure of a certain type at a lower level of abstraction can result in a failure of a different type at the higher level of abstraction [Cri91]. A failure behaviour can be classified only with respect to a certain specification, at a certain level of abstraction [Cri94].

Imposing structure is the basis for controlling complexity and hence is the basis of methodologies for designing and constructing both hardware and software systems.

### System Model

We have so far characterised systems in terms of their interactions with their environment, but we must address the issue of *what a system is* by stipulating the way in which systems are built up from their constituent parts:

- A **system** is defined to consist of a set of components which interact under the control of a design [LA90].

A *component* of a system is yet another system[1]. Thus, a system contains a set of component sub-systems, which cooperate so that their composed activity generates the external behaviour desired of the system. The external behaviour of a system is therefore the manifestation of internal activity within the system. The *internal state* of a system is defined to be the ordered set of the external states of its components; the *external state* of a system is viewed simply as an abstraction of its internal states.

In general, the design must ensure that each component receives as input an appropriate subset of the outputs of all the other components. Furthermore, the design is responsible for channelling system input to their components, and also for generating the system output as an abstraction of the component outputs.

As an example, the alternating bit protocol [BSW69], extended with timers, is a simple way to achieve communication over a medium that may lose messages. Consider the duplex communication medium of Figure 2.1, where $A$ and $M$ are media with potential faults[2]. We now consider, in turn, each concept set out before and discuss in terms of our example.

The *system*, *Sys* in Figure 2.1 is indicated by the outer rectangular box which has as its *interface* communication ports *in* and *out* for input from and output to its environment respectively. The components of *Sys* are $S$, $R$, $A$ and $M$ collectively which each interact to provide the system's service. *Sys* is itself the *environment* to these sub-components.

[1] Giving recursive definitions is not for recursion's sake. Rather, the aim is to emphasise relativity with respect to the adopted viewpoint.

[2] We will discuss a *fault hypothesis* later.

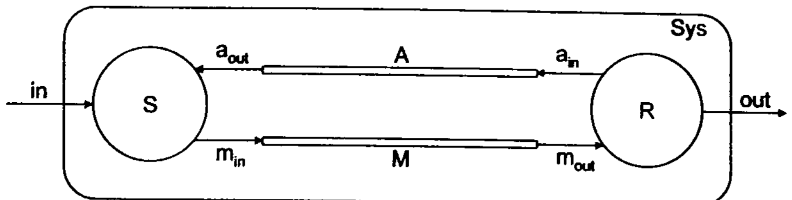



Figure 2.1: Duplex communication medium

### 2.1.2 Specification and Design

We first describe the notion of a system's *specification* and *design* and the relation between the two which then permits us to express the notion of erroneous transitions and states with a system and the specification thereof.

#### System Specification

*Specification*, according to [Bra03], is:

> *...the invention and definition of a behaviour of a solution system such that it will produce the required effects in the problem domain.*

The definitions of faults and failures must be given with respect to a specification of acceptable behaviour for a system. Given the fundamental role played by specifications, it is appropriate to describe the desirable properties of a specification. We focus on *timeliness* and *expected values*, typical properties used in the literature [ES85], which are appropriate to specifying component faults (c.f. Section 2.1.3). A *specification* for a system should be consistent, complete and authoritative, and can be applied as an effective test in all cases to determine the correctness of an implementation. The need for *consistency* in a specification is obvious. Further, a specification should be *complete* so that the behaviour of the system is defined for all possible inputs.

A specification of acceptable behaviour provides a standard against which the behaviour of the system can be judged:

- A **failure** of a system occurs when the behaviour of the system deviates from that required by its specification [LA90].

Clearly it is necessary to be able to stipulate what constitutes a deviation from specifications; it is only possible when the specification is complete and correct. The specification of a *black box* component is restricted to the externally visible events. Therefore, any deviation from a specification must be described using externally visible events.

### System Design

A design, according to [Bra03], is:

> *the decomposition of a system into its actual structural components for the purpose of constructing it.*

The design should be an informative system description, faithful to the specification, but avoiding irrelevant implementation detail. It should serve as a basis for discussions between the client, designer and implementor, and support objective analysis. Although identifying the structural components in a design, further implementation detail may be unavoidable given input from the customer. However, making such decisions in the design may lead to *design failures* which introduce, at an early stage, a fault in the design, which may lead to a failure, which is not detected until unit or system testing.

### Casual Analysis of system failure

Having defined a system in terms of its external and internal behaviour, we will adopt the work of Lee and Anderson [LA90] in analysing and naming the causes of system failures. Recall that when the system specification is authoritative, it need not be challenged and therefore the examination of the causes of system failures need only concern the internal operations of the system, i.e. the composition of the internal components. From our example (shown in Figure 2.1) the system *Sys* is made up of four components *S*, *R*, *A*, and *M* which interact under the control of design *D*. Let the external state transition of *Sys* from $E_1$ to $E_2$ constitute correct behaviour and be caused by the internal state transitions through the states $s_1, s_2, \ldots, s_{i-1}, s_i, \ldots, s_n$.

Suppose that the system fails by changing its external state from $E_1$ to $E_2'$ instead of $E_2$. Two concepts are needed to discuss the causes of this failure: first, an event has occurred within the system which should not have occurred; secondly, the occurrence of that event gave rise to a condition or state which should not have arisen. If the system had moved through the states $s_1, s_2, \ldots, s_{i-1}, s_i, \ldots, s_n$, the failure would not have happened. Therefore, at some stage, the internal state transition must have diverged from that sequence. Let us say that the transition from $s_{i-1}$ proceeded to $s_i'$ instead of going to $s_i$. Thus, the transition from $s_{i-1}$ to $s_i'$ is the first step towards system failure, and will be called an erroneous transition. Let the transition from $E_1$ to $E_2'$ be caused by the internal state transition $s_1, s_2, \ldots, s_{i-1}, s_i', \ldots, s_n'$; the states $s_i', \ldots, s_n'$ which are generated after the occurrence of the erroneous transition are termed as erroneous states.

### Erroneous Transitions and States

In the above scenario within *Sys*, the (erroneous) transition from $s_{i-1}$ to $s_i'$ was considered to have caused the system failure. However, it need not be the case that such a divergence from the intended sequence always results in a system failure. Consider, for example, that the transition from $s_{i-1}$ to $s_i'$ refers to flipping of a bit in communication channel, *A* from zero to one. (Occurrence of this event is clearly a failure of the communication component within *Sys*.) *Sys* will not fail, so long as the affected transmission (which is in an erroneous state) is not used during any computation, or if the bit is flipped back, or if the transmission is

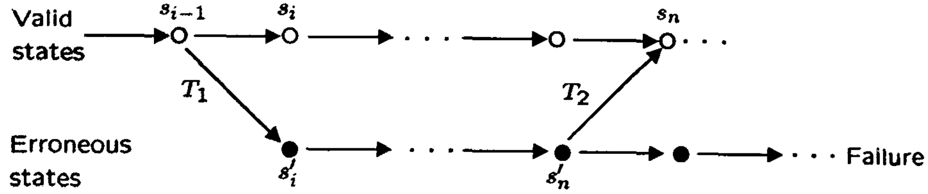



Figure 2.2: Erroneous Transitions and Erroneous States

repeated subsequently. Accounting for the possibility that a failure may not follow an erroneous transition, [LA90] carefully define erroneous transitions and states in the following manner:

Two essential concepts are required in detailing the causes of failure; an event which should not have occurred and a state which should not have been reached. These are termed *erroneous transitions* and *erroneous states* respectively:

- An **erroneous transition** of a system is an internal state transition to which a subsequent failure could be attributed. Specifically, there must exist a possible sequence of interaction which would, in the absence of corrective action from the system, lead to a system failure attributable to the erroneous transition.
- An **erroneous state** of a system is an internal state which could lead to a failure by a sequence of valid transitions. Specifically, there must exist a possible sequence of interactions which would, in the presence of erroneous transitions, lead from the erroneous state to a system failure [LA90].

The role of these definitions is illustrated in Figure 2.2, where arrows represent possible transitions between internal states in a system. All transitions are assumed correct with the exception of $T_1$. $T_1$ is an erroneous transition which potentially leads to a failure, in which case $T_2$ is representative of some corrective action which returns the system to a valid state. This illustration fits with the definitions given above, in that, not all faults ultimately lead to failure.

The descriptions above are intended to reflect the situation where, after a system failure, the history of a system is used to identify the cause of the failure. We later appeal to these definitions whilst defining our fault semantics.

Having identified an erroneous transition as the cause of a (possible) failure, we will continue to examine the causes of an erroneous transition. To simplify the analysis, we define the erroneous part of an erroneous system state as an error.

### Errors and Faults

It is common practice to use the terms error or fault for a specific defect within a system:

- an **error** is part of an erroneous state which constitutes a difference from a valid state

An error in a component or the design of a system will be referred to as a **fault** in the system:

- A **component fault** in a system is the result of a failure in the internal state of a component.
- A **design fault** in a system is a mis-representation in the design[3].

Following the terminology developed in [LA90], we will say that a component **fails** when its behaviour deviates from that specified. The term **fault** will be used to refer to the cause of the failure.

Let us continue the scenario from above, where the system *Sys* has failed due to the erroneous transition from $s_{i-1}$ to $s_i'$ (instead of $s_i$). Let the valid state $s_i$ be $\{e_S, e_R, e_A, e_M\}$ where $e_S$, $e_R$, $e_A$ and $e_M$ are the external states of components *S*, *R*, *A*, and *M* respectively; if the erroneous $s_i'$ is $\{e_S, e_R, e_A', e_M\}$, then $e_A'$ is the error in $s_i'$. That is, the external state of component *A* makes the system be in error. There are two cases: either *A* did not fail or failed during the erroneous transition. If *A* did not fail then no component failed within the failed system. In that case, the blame for the system failure must go to the system design whose role is to ensure that all internal state transitions are valid when all components' behaviour conforms to their respective specifications. An example of a design failure is an application program invoking unspecified operations on objects or permitted operations with incorrect parameters; in case of hardware subsystems, a design failure could be a missing or wrong connection which prevents proper interaction between components. Considered as a system which has failed, the design of *Sys* must contain an error, i.e., *D* must be in an erroneous state. (Recall that a system failure cannot be caused by anything other than an erroneous transition occurred within the system.)

Let us suppose that *A* failed during the erroneous transition within *Sys*. The analysis we have carried out so far regarding the failure of *Sys* can be applied (recursively) by considering *A* as a system in its own right. Let us denote by $e_a$ the error within *A* which caused *A* to fail. If *A* is made up of components $a_1$, $a_2$ and $a_3$ which interact under the design $d_a$, then the error $e_a$ is caused by the failures of one or more components of *A* and/or the failure of $d_a$. Let us say that only $a_1$ in *A* has failed. The cause of $a_1$'s failure can be traced to error(s) within $a_1$, caused by failures of $a_1$'s components and/or design. Thus, the ultimate cause of the failure of *Sys* can be pursued as far as considered worthwhile, or until atomic components are reached.

### Component/Design Failures

The distinction between a component and a design fault is apparent when they are exemplified. Failures of any system are due to a design fault or are the results of a component failure. The failure of a component is in turn attributed either to a design fault with the component or to a failure of a sub-component. Eventually, at some level, the original system failure will be attributed to a design fault, unless a failure of a component which is considered to be atomic[4] is held responsible.

However, should all components meet their specifications then the problem must lie in the design of the system. A specification of behaviour for the design of a system is that is should ensure all internal state

[3]Designs (in RTN-SL) are considered to be the *architecture* of the system. In other words, designs identify the components or elements of the system and their inter-relationship. Designs do not have states *per se*. Design faults are defects in the system architecture such that it is impossible to write software with that architecture which does not contain defects that result in errors and failures (e.g. too much –or not enough– concurrency or non-determinism; or not a connection path, etc.).

[4]The implication that any further internal structure cannot be discerned, or is not of interest, and therefore, can be ignored.

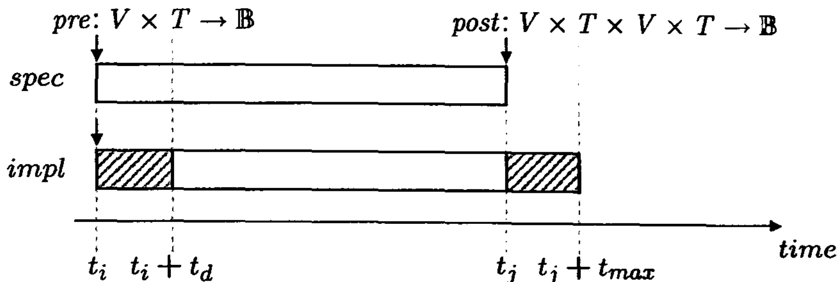



Figure 2.3: Specification and Implementation Model

transitions of the system are valid in the absence of any component failures. If there is an erroneous transition and no component has failed then the design of the system must have failed. Since the state of a design does not usually change, then such faults are considered permanent.

### 2.1.3 Fault Classification

Conceding no implementation can be perfect, we illustrate in Figure 2.3 the regions which are *acceptable* behaviours with respect to a specification. Informally, the response of a component for a given input sequence will said to be correct if the output value is not only as expected, but also produced on time. Figure 2.3 is more formally stated in Definition 2.1 where we define the *correct* (or acceptable) behaviour of a component from which we can then define faults

**Definition 2.1 (Correct Behaviour)** *Let a component receive an input at time $t_i$ and as a result produce an output value $v_j$ at time $t_j$. For that input, the response $v_j$ at time $t_j$ is correct iff:*

1. *$v_j = w_j$, where $w_j$ is the expected value consistent with the specification at time $t_j$, and*
2. *$t_j = t_i + t_d + \Delta t$, where $t_d$ is the minimum delay time of the component and $\Delta t$ is the unpredictable delay time such that $0 \leq \Delta t \leq t_{max}$ and $t_{max}$ is the maximum unpredictable delay time of the component.*

The values $t_d$ and $t_{max}$ are constants for a given component. First, we note that the implementation's response of a component is not assumed to be instantaneous for a given input but experiences a finite minimum amount of delay which is specified by $t_d$. Second, it is usual to indicate a *time interval* during which a response is expected, i.e. $t_i \longrightarrow t_j$.

Definition 2.1 gave the specification of a correctly operating component with respect to our model. We have allowed for acceptable response delay in an implementation of a component and illustrated the relationship between a specification and implementation in Figure 2.3. Figure 2.3 shows more than the expected timings of input and output of a component. The predicates *pre* and *post* specify the validity of a value. $V$ at a specified time, $T$. An acceptable implementation is also shown, which illustrates the acceptable response times. This figure illustrates the distinction between a *specification* and *implementation*.

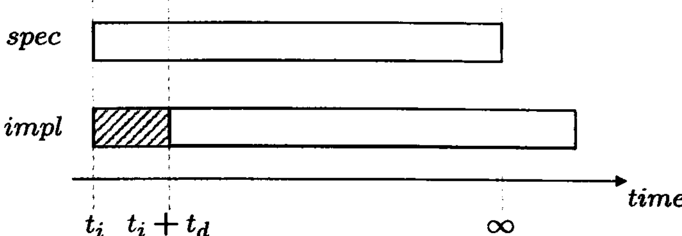



Figure 2.4: Omission Fault *Observable* Intervals

Faults are the primary causes of system failures. We present a fault and failure classification for components using 'timeliness' and 'expected values' as the two distinguishable properties of a component's response.

A reliable computing system should be capable of providing guaranteed services in the presence of a finite number of component failures. In order to be able to provide any kind of guarantee of services, the system designer must specify what kind of, and how many, component failures the system is intended to tolerate.

### Abstract Fault Definitions

Ezhilchelvan and Shrivastava [ES85] classified faults with respect to their (failure) consequences. A fault that results in system output not being produced at all is termed an *omission* fault. If a correct output is produced but at the wrong time this is said to be a *timing* fault. If an incorrect output is produced at the proper time, the fault is said to be a *value* fault.

**Definition 2.2** ***Omission Fault:*** *A fault that causes a component not to respond to a nonempty input sequence will be termed an* omission *fault:*

1. $v_j = null$, $t_j = t_i + t_d + \Delta t$ *and*
2. $v_j = w_j$, $t_j = \infty$

Figure 2.4 shows that an omission fault must occur at an *absolute* final time. Ideally this value would be $\infty$ but in a computing environment this is not possible and we must choose a suitable value. Giving a suitable value for $\infty$ is made easier by defining a *window of observation*. That is, each fault classification is observable with respect to a correct specification for a fixed, or well defined, period as shown in Figure 2.5. For example, consider an omission fault. One can accept that the *ith* occurrence of an event is *missing*, presumed lost, once the $i+1$ occurrence appears.

**Definition 2.3** ***Timing Fault:*** *A fault that causes a component to produce the expected value for a given nonempty input sequence either too early or too late will be termed a* timing *fault:*

1. $v_j = w_j$ *and*
2. *either* $t_j < t_i$ *or* $t_j > t_i + t_d + t_{max}$

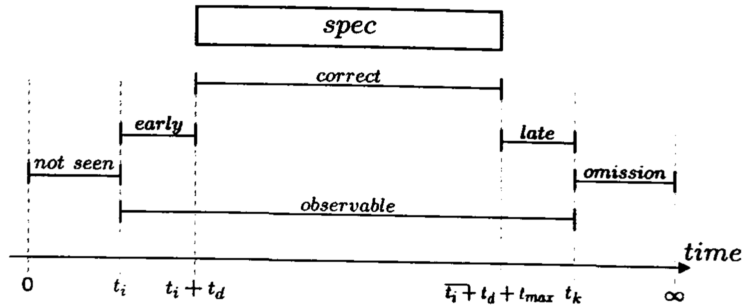



Figure 2.5: Observable fault intervals

**Definition 2.4** ***Commission Fault:*** *When a component produces a value that is not expected, then a* commission *fault has occurred.*

1. $v_j \neq w_j$ *and* $t_j = t_i + t_d + \Delta t$ *or*

2. $v_j = w_j$ *and* $t_j \neq t_i + t_d + \Delta t$ *or*

3. $v_j \neq w_j$ *and* $t_j \neq t_i + t_d + \Delta t$

**Definition 2.5** ***Value Fault:*** *When a component that produces an output at an anticipated time which does not satisfy the specification, then a* value *fault has occured.*

1. $v_j \neq w_j$ *and* $t_j = t_i + t_d + \Delta t$

Referring to Figure 2.5, the time markings, $t_i$, $t_d$, $t_{max}$ and $\infty$ are used from the definitions given previously, where $t_j = t_i + t_d + \Delta t$, $0 \leq \Delta t \leq t_{max}$ and $t_i + t_d + t_{max} \leq t_k \leq \infty$ where $t_k$ is a discernible value. We later suggest values of $t_k$ specific to our architecture.

## 2.2 Fault Hypothesis

The precise definition of any assumptions of the types of faults, the rate at which a component fails and how they fail, are an essential step in the design of a fault tolerant system. Yet, at the earliest stages of design of new software, nothing is known about its failure modes, nor their effect on a system's behaviour. However, it is at this stage that cost effective measures to deal with such failure modes is important. The assumed type of nature of faults indicates the type of redundancy that must be implemented within the system.

The term *fault hypothesis* is standard in the literature [Cri85], yet the term is mis-leading. Rather, we intuitively mean **error hypothesis** or even a **failure hypothesis**. Consider as an example a radar sub-system. We expect as an output a periodic sample which is a snapshot view of some airspace. Then considering the detrimental outputs possible, such as no output, an output which is later than the known

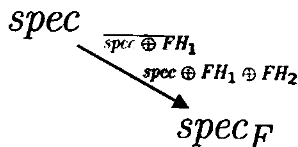



Figure 2.6: Weakening the specification

period or an incorrect output, are they faults, errors or failures of the radar sub-system? The inconsistency between the definition of faults, errors and failures (c.f. Section 2.1.3) and the term *fault, error* or *failure hypothesis* is due to the perception, in this case, of the radar sub-system - is it an atomic component or a system of some description? In reality, we are interested in the failure modes of the radar (sub-)system, not the faults as cause too, but the observation of a detrimental output.

We therefore require a hypothetical process which challenges each design component to outline the potential failure modes of a design. In the absence of accurate and concrete statistical evidence, we are forced into this hypothetical process, which is an investigation of 'what if' questions of each component of a design. On finding a negative response, we then challenge the design for the effects of the fault to determine whether the failure mode is critical. If the outcome is negative –that the fault is critical– then we must incorporate the fault definition into our specification. Shown in Figure 2.6 is the relation between an original (non-faulty) specification and the incorporation of several fault definitions, or *fault hypotheses* (FH), to define the (weaker) specification $spec_F$. The relation $spec \oplus FH_n$ indicates the disjunction of the behaviours specified in the original specification or those in a fault-hypothesis are observable for the design. It is therefore the requirement to consider these behaviours against any safety properties the design must uphold and guarantee the satisfaction of these properties given (only) the possibility of faulty behaviour specified by each $FH_n$.

The complete definition of a components fault hypothesis entails an assertion on errors occurring in all domains (time and value).

## 2.3 Fault-Tolerance

The aim of fault tolerance is to prevent a system from failing and the discussions in the previous section indicate that meeting this aim requires the use of techniques to detect and correct errors before a failure could occur. The fault tolerance techniques used within a system are inevitably linked to, and influenced by, the design and architecture of that system. This means that there cannot possibly exist a general technique for implementing fault tolerance. This section presents the general principles identified by [LA90] which underly all known fault tolerant systems, and then examines certain ways by which these principles are implemented in practice.

### 2.3.1 Principles of Fault Tolerance

Given the distinction between *faults* and *errors* above, and the relationship:

$$\text{fault} \rightarrow \text{error} \rightarrow \text{failure}$$

then when there is a fault in the system, this fault can lead to errors in the state of a system, which -may- subsequently lead to a system failure. The aim therefore of fault tolerance is to prevent this causal relation of errors and faults, and we therefore require techniques for detecting and treating errors and faults.

### Redundancy

All design techniques that provide fault tolerant strategies are based upon *redundancy*. Redundancy involves duplicate components to perform the same, or similar task within a system. Often redundant components can be grouped together as a sub-system which satisfies the specification of a (original) single component. The arrangement of such components is determined by the redundancy strategy and detailed in a system design. Often these additional components are infrequently executed, with only a *primary* component satisfying the specification, hence the term redundant. The additional components are necessary, and intended for, the sporadic or unusual data input.

Redundancy strategies can be categorised as either *static* or *dynamic*. A static redundant strategy would arrange the components such that any faults or errors from a component are masked from the environment of the system. However, a dynamic strategy would provide detection mechanisms for which redundancy elsewhere would achieve the fault tolerance.

### Example replication strategy

A standard example of the use of static redundancy is *Passive State Replication* (PSR). Typical applications of this strategy are to provide tolerance against timing and omission faults. To tolerate a fault of -say- component $E$ in Figure 2.7(a), $E$ is replaced by three passively replicated components illustrated in Figure 2.7(b) which satisfy the same specification as that which $E$ did. By passive, the intention is that each component is executed in turn until an acceptance test is successful. If the acceptance test for $E_{rp1}$ fails, then $E_{rp2}$ is executed. $E_{rp3}$ cannot fail its acceptance test.

Given the specification of the component(s) we are transforming, we know that condition $c_1$ must hold on entry to $E$ and that condition $c_2$ must be satisfied on the exit from $E$. Therefore, we must assume condition $c_1$ is true in the transformed design and we must guarantee, in addition to our real-time requirements, that condition $c_2$ also holds on exit. Incidentally, $c_2$ becomes the *acceptance test* and therefore the exit condition from each replica. With regards the *failure test* we must ensure that all the exit conditions form a tautology when specifying the fault hypothesis as the failure test.

The semantics of RTN-SL (presented in full in Section 2.4) state the exit condition of $E$ is more than just condition $c_2$. Additionally, the timing specification of $E$, the *best case execution time* (BCET) and the *worst case execution time* (WCET) should be respected. We therefore illustrate these semantics and a possible fault hypothesis in our example, that the fault hypothesis states a *late fault*. That is, an exit condition to $E_{rp1}$ and $E_{rp2}$ states "$E_{rp1}$ *transitions to* $E_{rp2}$" and "$E_{rp2}$ *transitions to* $E_{rp3}$" should a late fault be observed at $E_{rp1}$ and $E_{rp2}$ respectively. Then, $c_2$ and FH should form a tautology.

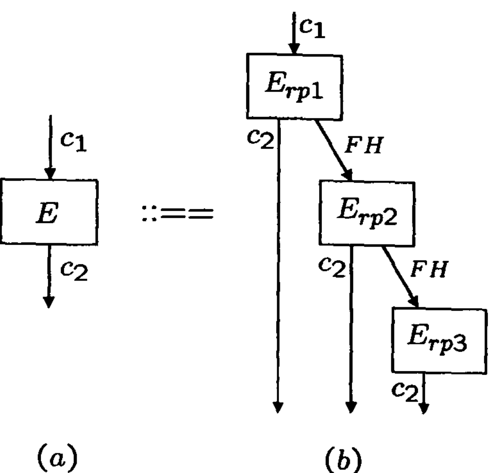



Figure 2.7: Passive State Replication

We later consider another replication strategy, called Triple Modular Redundancy (TMR) [WLG+78], however the focus of this thesis is not to formalise a number of fault tolerant strategies, but to provide for the design methodology which makes applying such strategies possible for real-time systems.

### 2.3.2 Formalisms of Fault Tolerance

In the literature on the formalisation of fault tolerance, the earlier works [LJ96, Cri94, AK98b] do not explicitly model the occurrence of faults. Rather, much of the earlier work concentrated on process algebras, particularly CCS [Pra84], and proved that the behaviour of a fault tolerant system is equivalent to that of the corresponding fault free system. In [Pra84], it was assumed a failure is detected by an acceptance test either by a faulty process itself or by a communicant. Further, that should a process detect itself to be faulty, then it should stop *silently*, until it is remade. Fail-stop processes were first formally defined by Schlichting & Schneider [SS83], who also considered the problem of implementing them. They therefore just assumed that failures, defined as derivations from the specified "normal" behaviour, are detected by some means at some points after they occur. This was sufficient for showing a fault tolerant design *observationally equivalent* to the original design, but does not allow one to determine and derive a fault strategy for a non-fault tolerant design should the fault tolerant design not yet exist.

Schepers and Hooman [SH94] use trace-based equivalences for proving fault tolerance. Faults are not treated as special actions; only the effects of faults on the externally visible input and output behaviours of the system are modelled. In [Nor92], Nordahl applies CSP and trace theory to develop a fault tolerant system design. The system is a collection of processes. A fault tolerant technique is realised by a combinator operator which can be one of the CSP operators, a composition of these operators, or defined in terms of triples.

Lamport and Merz [LM94] use a calculus for fault tolerance analysis based on the temporal logic of actions

(TLA) [Lam94]. In this work it is shown how the TLA can be applied to specify and verify fault tolerance algorithms. A specification is a mathematical formula and theorems asserted in the specification are proved using the method of structured proofs. This allows one to hierarchically structure proofs, where simple proofs can be discharged mechanically through the TLP verification system [Eng95]. Liu and Joseph [LJ96, LJ99] similarly use TLA to reason formally about the properties of an action system. The work presents a transformational framework in which it is assumed that the physical faults of a system are modelled as being caused by a set $F$ of *fault operations* which perform state transformations in the same way as the ordinary program operators.

The availability of automatic tools for verification also makes a framework usable in an industrial context. For example, the successful approach in the European Project GUARDS (Generic Upgradeable Architecture for Real Time Dependable Systems) [PABD+99] to validate two basic fault tolerance mechanisms: the inter-channel consistency network mechanism [BFG99] and the fault treatment mechanism [PABD+99] illustrated how large scale problems can be tackled

Kulkarni et al [KRS99] argue that the decomposition of a fault tolerant program into its components is beneficial in its mechanical verification, and that such a decomposition admits reuse of the proofs for other fault tolerant programs as well as the variations of the given fault tolerant program.

A common way of showing a fault tolerant design equivalent in the presence of faults to a non-fault tolerant design is to show *observational equivalence*. Observational equivalence, first introduced in [Mil80], is based on the idea that the behaviour of the system is determined by the way it interacts with the environment: two systems are equivalent whenever no observation can distinguish them.

### Conclusions

Explicitly modelling the behaviour of faults is essential if we wish to reason about faults in a real-time context. However, we feel that the effects are more crucial than the cause, and choose to model the *observational* effects of faults on design components. Only concerning ourselves with the observational effects, we allow for internal state transitions of components which are erroneous, but do not lead to failure, not to interfere with our modelling. Rather, by recording the observation of a fault as a *fault event* in a trace model, we provide for a history of a RTN, over which we can reason about faults, and more crucially, fault tolerance. Our intention to introduce new operations, or actions, into RTNs that generate fault events which are distinct to the existing event model suggests a reactive approach to fault tolerance: given the observation of a fault event then *action* a fault tolerant operation. Therefore, those faults we do not consider are still undetected and may lead to non-deterministic behaviour. Finally, we can show a fault tolerant RTN *observationally equivalent* to its non-fault tolerant counterpart should the non-fault tolerant trace be a subset of the fault tolerant design, excluding the fault events.

## 2.4 Real-Time Networks

In Chapter 1, we alluded to scale and complexity as common characteristics of real-time systems. A proven method for dealing with any complex system is to partition it into smaller independently operating subsystems which only interact with one another and with the system environment through explicitly defined

communication connections. For real-time systems, one approach is the Real-Time Network (RTN) architecture which does partition concurrent processing components and enforces communication for data exchanges and synchronisation through shared data via explicit connections.

MASCOT (Modular Approach to Software, Construction, Operation and Test) is a design methodology based on the real-time network concept [Sim86]. It comprises a design language and graphical notation, together with a process for design derivation based on structural decomposition. This involves identification of computational components of a system (its subsystems and processes) and the interactions between these components (i.e. data-flow paths), together with protocols that characterise these interactions. MASCOT-3 has been advocated for the design of large concurrent or distributed, real-time embedded software systems and has been used extensively throughout the defence industry [Woo96].

DORIS [Sim94], the Data-Oriented Requirements Implementation Scheme is a variant of MASCOT-3 developed by MBDA. The main difference is that DORIS distinguishes three levels of design abstraction, such that application network designs and execution network designs are specified in addition to the functional design. The former define the logical architecture of a system, whilst the latter extend the application network to include all additional components required to support distribution across a particular processor network. This enables flexible re-mapping of a design to the hardware platform as the hardware platform evolves [PAH00]. DORIS also supports a wider-range of synchronous and asynchronous communication protocols appropriate to both shared-memory and message passing implementations [Sim03].

In this thesis, we focus on support for transforming designs represented using RTN-SL, a specification and design language (intended for integration into MASCOT-3 and DORIS) being developed by MBDA for defining the behaviour of real-time networks [Pay02]. The following sub-section provides an overview of the RTN-SL and its graphical notation in particular.

### 2.4.1 RTN-SL

RTN-SL is used to define flat Real-Time Networks, including the real-time and functional behaviour of their activities. A flat RTN comprises a set of activities (single-threaded processes) which are connected into a network by ports that interface with communication paths. Between each activity is an intercommunication data area (IDA) which defines the interaction protocol used on that path. Activities may not be directly connected together, but must communicate via an IDA [Pay02].

The RTN-SL has both a graphical and textual representation where the latter is a precise specification of the functionality of the graphical form. The concrete textual syntax of the RTN-SL is influenced by a number of languages, particularly by Ada [Bar96], VDM-SL [LHP$^+$96, FL98] and the PVS logic [OSRSC99a]. The concrete textual syntax is not introduced here and the reader is referred to [Pay02] for a full reference.

RTN-SL supports five standard protocols[5], namely pool, channel, signal, stimulus and dataless channel; the three basic protocols are the pool (similar to a shared variable), channel (a bounded buffer) and signal (a one-place over-writing buffer). These impose different synchronisation constraints on the reader and writer of the protocol depending on whether they allow data to be destroyed when it is read or written (see Table

[5] Several variants of these standard protocols where added to the language in [Pay02], such as *Flash Data*, *Prod* and *Overwriting Buffer*, however the fault tolerant strategies suggested in this thesis only use the standard protocols and the reader is referred again to [Pay02] for a full reference.

| | Non-Destructive Read<br>Read data not consumed<br>(Never held up) | Destructive Read<br>Read data consumed<br>(Held up when no data) |
|---|---|---|
| Destructive Write<br>Old data overwritten<br>(Never held up) | Pool | Signal |
| Non-Destructive Write<br>Old data not overwritten<br>(Held up when no space) | Constant | Channel |

Table 2.1: The Basic Protocols

2.1 for a summary). The two other RTN-SL protocols, namely stimulus and dataless channel, are variants of signal and channel respectively; they differ in that both allow communication of void (null) data.

Abstract data types (ADTs) may be defined to support definition of a flat Real-Time Network. These are necessary when a data type must be visible in more than one activity and/or IDA where one activity communicates data to another which is not a built-in type [Pay02].

Additionally, the RTN-SL includes a sub-language known as Activity Description Language (ADL) for defining the behaviour of activities. In turn, the ADL includes a timed state-machine notation sub-language termed the Activity State-Machine (ASM) which is used to define the structure and timing constraints of an activity's algorithm.

ASM distinguishes between static and dynamic states; static states model synchronisation points of an algorithm. For example, when an activity is in a static state, it maybe attempting to communicate using a synchronous protocol. Transitions from a static state are labelled either with the event indicating that the communication may continue or finish, or with the lower and upper bounds of a time delay [Pay02].

Dynamic states model an activity's computation, each one encapsulating some non-reactive functionality. They have a best-case execution time (BCET) bound, a worst-case execution time (WCET) bound and worst-case response time (WCRT) bound; an optional worst-case response time on read may also be recorded. An activity is *normally* required to exit a state within these times. Transitions from a dynamic state are labelled with conditions over the local activity's state which are evaluated when a dynamic state terminates [Pay02]. A composite dynamic state comprises several dynamic states whose operations may execute non-deterministically.

Usually, the RTN-SL graphical syntax, which builds upon the graphical syntax of MASCOT [Sim86], is commonly used when discussing RTN-SL specifications. Activities are represented as circles; IDAs by the appropriate symbol from Table 2.1 (which is enclosed within a rectangle where the IDA is defined and hence named); ADTs by diamonds; communication paths by solid arcs with arrows indicating the direction of data-flow; and ADT imports by dotted lines with arrows pointing towards the unit that imports the ADT.

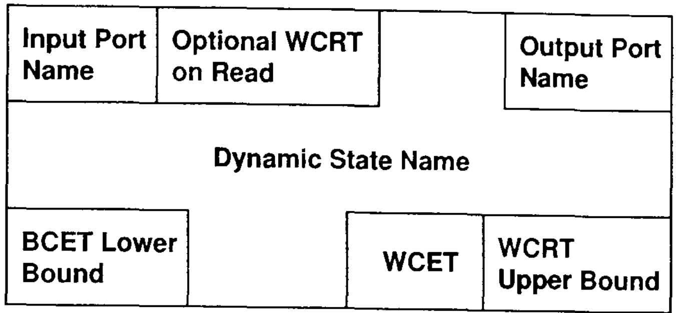



Figure 2.8: Dynamic State

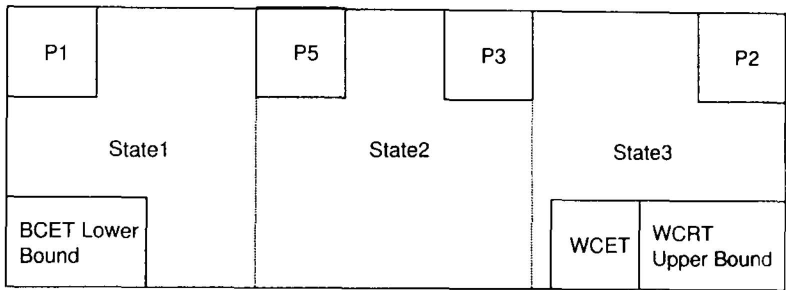



Figure 2.9: Composite Dynamic State

Ports are represented by solid circles on the perimeter of activities and the state machine drawn within the activity circle.

Static ASM states are represented by an ellipse and dynamic states as a rectangle. Where an input or output is associated with a dynamic state, then the appropriate port name is shown in the top left hand and right hand corners of the rectangle respectively. Labels in the bottom left and right hand corners indicate BCET, WCET and WCRT; the optional WCRT on read is shown in a box next to the input port name. The dynamic state graphical syntax is summarised in Figure 2.8.

Composite dynamic states are depicted by partitioning a dynamic state using dotted lines. Each constituent state may have its own input and output port, although one set of time bounds applies to the whole state. An example composite comprising three dynamic sub-states is shown in Figure 2.9.

Transitions between states are depicted by directed arcs pointing from the source to the target state. Conditions events and time-outs are shown as textual labels adjacent to the appropriate transition. The initial state is shown by being the target of a transition with no source.

Figure 2.10 illustrates the main constituents of an RTN-SL specification as described above. Activity *A2* includes three static states, *A* (the initial state), *C* and *E*, together with two dynamic states, *B* and *D*. When the activity is in state *A* and a void data value is present in the stimulus connected to port 2 (*p*2), then event

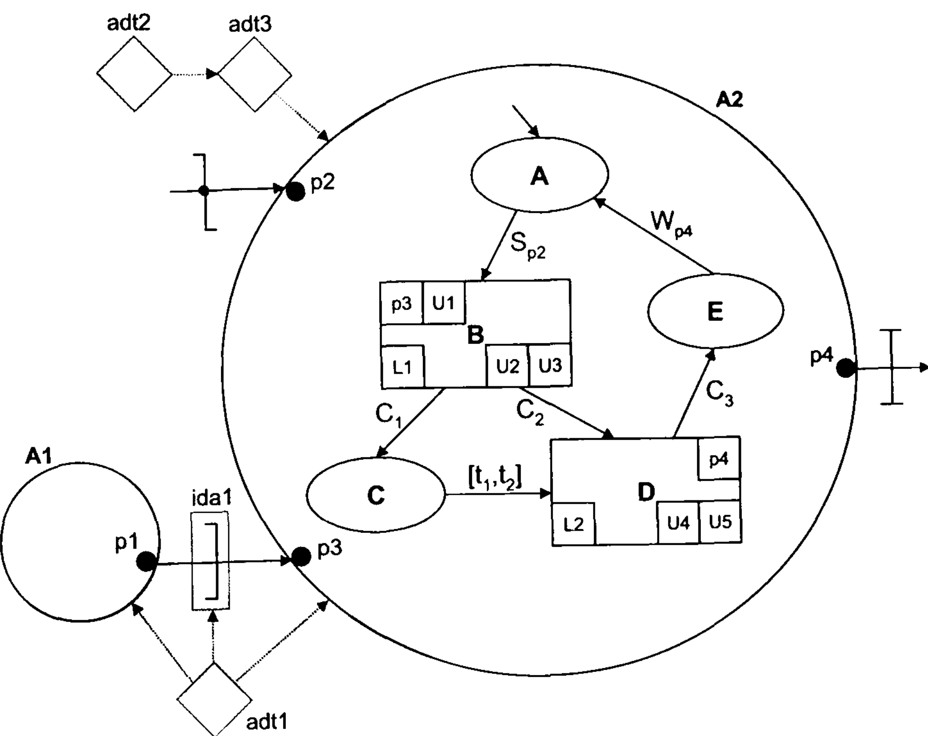



Figure 2.10: An Example Graphical RTN-SL Network

$$D \xrightarrow{\mathit{NetSpec}} \Omega[\![D]\!]$$

Figure 2.11: Existing Semantic Framework

$S_{p2}$ occurs and the activity moves to state $B$ which reads from the pool on port 3 ($p3$). The computation (not given here) associated with state $B$ is executed and takes between $L1$ and $U3$ time units; conditions $C_1$ and $C_2$ are then evaluated. A transition with a true condition is taken and either state $C$ or $D$ is entered. Assuming state $C$, the activity is de-scheduled for between $t_1$ and $t_2$ time units whereupon state $D$ is entered. The algorithm associated with state $D$ (again not given) is executed between its time bounds (as state $B$) and a value written to the channel linked to port 4 ($p4$). The activity waits in state $E$ until the $W_{p4}$ event occurs whereupon it returns to the initial state ($A$). Activity $A1$ executes concurrently with $A2$, communicating with it via IDA *ida*1. ADT *adt*1 is used in the definition of $A1$, $A2$ and *ida*1; ADT *adt2* is used in the definition of *adt*3, which in turn is used in the definition of $A2$.

## 2.4.2 Verification of Real-Time Networks

Given a real-time specification language which provides a formal framework for top-down design of real-time systems, one requires tool support for the ensuing verification effort. To obtain mechanised support for our chosen formal framework, the semantics of RTNs are represented as a *shallow embedding* [OSRSC99a] in the PVS logic, which is a typed higher-order classical logic and supported by the PVS (Prototype Verification System) [OSRSC99b] verification system.

The principal tool which defines this *embedding* is NetSpec [Pay01d]. NetSpec first syntax and type checks an RTN-SL specification ($D$) file that conforms to the notations static semantics, NetSpec then –with the correct switch– instantiates the axiom schema of $\Omega$, the RTN-SL semantic function, to produce a set of axioms which encode the behaviour of the specification (Figure 2.11).

The generated PVS theory from NetSpec naturally type checks with respect to the PVS type-theory. However, the theory section of an RTN-SL specification is *free hand* and may generate errors in PVS. Once we have the theorems we wish to prove, for example timeliness or safety properties, we begin to reason with our axiomatic semantics within the PVS system. Some automation of trivial goals are completed by PVS, but the bulk of the effort is driven by the designer. We later appeal to our transformational approach as aiding the designer in this effort.

Earlier experiences with this proof approach were successful. In [PAH00], a small example –principally the ASM– proved two theorems of liveliness and safety. The fact that both theorems were proved from the theory obtained by applying $\Omega$ provides some validation of the adequacy of the semantics and approach. Their experience supported the need to plan formal proofs before embarking on using a prover, and that formal proofs need to be reconverted to rigorous proofs for presentation and comprehension. Also, that the identification of suitable lemmas improves the modularity and clarity of proof, which gives hope that we can benefit our transformational methodology again towards this proof style.

Other methods and approaches have been similarly advocated in the literature. Hooman [Hoo97] proposes

a compositional verification method, again utilising the PVS system, which enables the design to be a framework of specification and programming constructs mixed freely, to formalise intermediate design stages during the top-down design process. Liu and Joseph [LJ97] extend their work on formalisation of fault tolerance with TLA[6] to consider refinement steps in an real-time design, as a way of verifying the timing properties inherent in real-time systems. Their work considers the balance of a low or high level abstraction which is feasible to verify, yet recorded accurately, the timing properties.

## 2.5 Methodology

### 2.5.1 The SHARD Analysis Method

We describe a method which draws on techniques from the chemical industries Hazard and Operability (HAZOP) analysis [CIS77], combined with work on software failure classification to provide a structural approach to identifying the failure modes of software [MP94]. The Hazard and Operability (HAZOP) analysis system of *imaginative anticipation* of hazards provides precisely the type of structured analysis which can feed in to the development of a new system. A HAZOP study attempts to identify previously unconsidered failure modes by suggesting hypothetical faults for review. The software specific method was christened Software Hazard Analysis and Resolution in Design (SHARD) [MNPF95], to avoid confusion with *traditional* HAZOP and is based on the MASCOT design notation. In the SHARD approach, as each part of the system design is produced, an analysis is produced based on the principles of HAZOPs. This analysis must either be shown to justify the design proposal, or impose a number of emergent requirements which must be satisfied later in the design development.

The decision to use SHARD as the basis for our work favours the use of design notations which employ a structural model of the system of the system which partitions the system into independent processes and defines the interfaces between them. The fault transformations possible within a passive IDA are much more restricted than those possible within an activity, providing good fault containment properties and an effective basis for analysis. The method we adopt, may not itself be sufficient for a complete safety analysis argument, but does provide the initial analysis to those parts of the system where faults may prove critical.

Illustrated in Figure 2.12 is the relationship between the original specification, design and the derived fault tolerant design which addresses the fault definition found critical by a SHARD analysis. Figure 2.6 highlighted the intermediate steps to reach the faulty specification from the original, it is therefore the responsibility of a SHARD analysis to define clearly which component behaviours may lead to a critical fault so they can be removed, or tolerated in a transformed design. The iterative process continues until all critical faults have been addressed, which cumulates to our fault tolerant design, $D_{FT}$.

#### 2.5.1.1 Method

The basic unit for analysis of a software design is a single RTN specification representing a system or subsystem. This system or subsystem consists of components –activities and IDAs– connected by data flow paths. Separate tables are produced for each RTN component. The process begins with a top-level (context) RTN diagram of the system. The major steps of the method are outlined below:

[6] mentioned in Section 2.5.2.

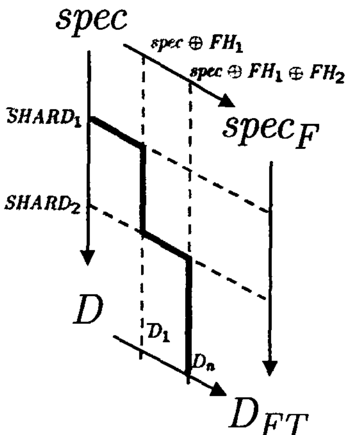



Figure 2.12: A SHARD invoked design process

- Each component and data flow are uniquely named;
- The design is reviewed to ensure it correctly models the intended system;
- A table of guide words is constructed;
- For each data flow in the system, each guide word is considered, with the evaluation recorded in a tabular format;
- The **potential causes** of each identified fault are determined;
- The **effects** of each hypothetical fault are considered and recorded;
- The set of hypothetical faults is reduced to a set of *meaningful* faults and the justification recorded.

#### 2.5.1.2 Guide Words

A considerable amount of research has been carried out into the classification of software failures [BS90, ES85], for which the categorisations are based on a component, analogous to our model of information flow to and from components.

The correctness of a component is specified in terms of two parameters: the *value* associated with it, and the *time* at which this value is presented. The value domain is divided into four categories: *correctly valued, subtle incorrect, coarse incorrect & omission.* The time domain is similarly divided into four categories: *correctly timed, early, late & infinitely late*[7]. The summary of the possible combinations of time & value faults is show in Table 2.2.

Considered previously from [ES85] was a *commission* fault, which expressly includes events such as completely unexpected output, which is specifically considered in our analysis.

[7]The distinction between an omission in the value domain and infinitely late is assumed to be made by a *perfect observer*

| Time | Value | | | |
|---|---|---|---|---|
| | **Correctly Valued** | **Subtle Incorrect** | **Coarse Incorrect** | **Omission** |
| **Correctly Timed** | Correct Service | Undetectable failure | Detection on value syntax or semantics | Detection at $T_\infty$ |
| **Early** | Detection on time | Detection on time | Detection on value syntax or semantics and/or time | Detection at $T_\infty$ |
| **Late** | Detection on time | Detection on time | Detection on value syntax or semantics and/or time | Detection at $T_\infty$ |
| **Infinitely late** | Detection at $T_\infty$ | Detection at $T_\infty$ | Detection at $T_\infty$ | Detection at $T_\infty$ |

Table 2.2: Fault classes and detectability

We therefore consider that a complete set of suitable failure classes which represents the guide words used for the SHARD analysis:

**Service :** Omission
Commission
**Timing :** Early
Late
**Value :** Coarse Incorrect
Subtle Incorrect

These failure classes are then considered in the context of an RTN design, to mature the formal definitions of each. The failure categorisations and their context-sensitive meaning is shown in Table 2.3.

As noted in [MNPF95], the original intention of the guide words in Table 2.3 was to find the minimum set which we could be reasonably confident would prompt consideration of the plausible failure modes of software. Though McDermind et al suggest from their experience with their study, had the "raw" words (i.e. *omission, commission, early, late, ...*) been used, there would (probably) have been fewer difficulties with interpretation [MNPF95]. We therefore use the guide words interchangeabilly, choosing the most appropriate word to aid comprehension in a given context. For example, "unwanted updates" are appropriate when considering data components, however it is hard to derive a meaningful intuition of it for an Activity, whereas a "crash" guide word is more appropriate.

Table 2.4 shows a fragment of the analysis output of the design shown in Table 2.10. The column heading **M?** records whether a hypothetical failure mode has been identified as *meaningful*.

| | Failure Categorisation | | | | | |
|---|---|---|---|---|---|---|
| | Service Provision | | Timing | | Value | |
| Component | Omission | Commission | Early | Late | Subtle | Coarse |
| Pool | No Update | Unwanted Update | N/A | Old Data | Incorrect | N/A |
| Signal | No Data | Extra Data | Early | Late | Incorrect | Inconsistent |
| Channel | No Data | Extra Data | Early | Late | Incorrect in Range | Out of Range |
| Dynamic State | No Update | Unwanted Update | Early Exit | Late Exit | Incorrect in Range | Out of Range |
| Static State | N/A | N/A | Early Exit | Late Exit | N/A | N/A |
| Activity | Crash | N/A | Early | Late | N/A | N/A |

Table 2.3: Table of guide words applicable to RTNs

| | |
|---|---|
| **Drawing Ref** | 1 |
| **Drawing Name** | Top Level |
| **Flow ID** | p3 |
| **Protocol** | Pool |
| **Data Type** | adt1 |
| **Additional Information** | ... |

| Guide Word | Deviation | Causes | Detection | Co-effectors | Effects | M? | Justification / Design Proposals |
|---|---|---|---|---|---|---|---|
| Old Data | Old pool data read by state $B$ | Period of $p2$ | | $A1$ failed to update $ida1$ | Stagnant data used in computation & Repetition of data written to $p4$ | NO | **Justification:** By design to use a Pool |
| No Update | state $B$ fails to read $p3$ | Period of $p2$ | | None | Stagnant data used in computation | No | **Justification:** By design to use a Pool |
| Late Exit | state B exits later than $u_3$ | Late read @ $p2$ | | None | Late entry to state $D$ | Yes | **MAction:** aka state B tolerant to a timing fault |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | |

Table 2.4: A fragment of an SHARD analysis of the example network in 2.10.

### 2.5.2 Design Transformations

The idea of design transformations for fault tolerance is not new. Other approaches have been proposed to study fault tolerance for systems applying error detection and recovery. In particular, using program transformations, Liu and Joseph [LJ92] show that proof of fault tolerance is not different from the proof of any other functional property. Effort has also concentrated on automating the addition of fault tolerant solutions to an intolerant design [KA00].

The motivations for a transformational method are three-fold. The first motivation comes from the fact that the designer may have a fault-intolerant program and it is known to be correct in the absence of faults. The second motivation is that the use of such automated transformation will obviate the need for manually constructing the proof of correctness of the transformed fault tolerant program as the transformed program will be correct by construction. This is especially useful when designing concurrent and fault tolerant programs as it is well-understood that manually constructing proofs of correctness for such programs is especially hard.

The third motivation stems from previous work by Arora and Kulkarni [AK98a, AK98b] that shows that a fault tolerant program can be expressed as a composition of a fault-intolerant program and a set of 'fault tolerance components'. Such fault tolerant components are described in some template format (preferably one akin to RTNs [Bor98]) which details where and when a template is suitable and hopefully proof of correctness to show tolerance against a class of faults.

Arora and Kulkarni [AK98a] propose a method where components are added to a design in a stepwise fashion. Each step adds a component to the design which tolerates one fault class, until the design is tolerant to the full set of fault-classes. Components are either detectors or correctors [AK98b]. Intuitively, a detector *detects* whether some predicate is satisfied by the system state; and a corrector detects whether some predicate is satisfied by the system state and also *corrects* the system state in order to satisfy that predicate whenever the predicate is not satisfied. Decomposition of a fault tolerant program permits the verification of a given property by focusing on the component that is responsible for satisfying it. For example, if we need to show that a program eventually recovers to a state from where it satisfies its specification, we should focus on its *corrector* components. Similarly, if we are interested in showing the absence of faults, then we must focus on the *detector* components.

Other approaches have proposed redundancy management within software modules, namely fault tolerant real-time structures (FERTs) [BSS94]. The approach deals with the use of software implemented fault tolerance in those real-time systems where a designer wishes more run-time flexibility than that afforded with static redundancy. The method introduces a three-level framework which manages the functionality through to the scheduling of the components. Although this approach in not applicable for our problem, it does support the idea of composing a system with a fault tolerance component in a dynamic, transformational approach.

## 2.6 Conclusions

From this survey of existing work in the literature on "Faults, fault tolerance & fault treatment" we have found the scope and motivation to propose a design transformation methodology for RTNs to provide for

classic fault tolerant techniques.

It has been noted that faults have not been considered explicitly in RTNs, although much development has been made for other real-time design methodologies, we feel the RTN approach is capable of sustaining a study of this nature. The obvious benefits are expected to be gained by extending a relatively new, yet increasingly accepted, design methodology as our foundation. Working within the RTN-SL methodology, rather than proposing a bespoke one, allows the clear specification of faults with respect to an RTN. Further, having existing examples allows for the presentation of transforming a design, rather than contriving new ones and facilitates the evaluation of the transformation against existing practices. RTNs also provide clear and discernible characteristics with which to specify fault definitions, as to a tightly structured graphical syntax.

In addition to the control and structuring offered by a transformational design method, we require structuring support for the proof task. Specifically, the form of the transformation –the arrangements of components being removed– should suggest the properties which are presented. For example, consider a component is removed which reads a value, computes some result and writes that result to an actuator within a time bound. Should we consider a value fault, then the new arrangement of components should still read and write a result within the time bounds, in addition to computing an agreed value. We therefore would consider this timeliness guarantee to be suggested by the transformational methodology.

Finally, we have identified a structured approach to challenging designs to derive the faults feasible, and more importantly, the critical effect upon the overall success of the system at a component level. This then defines our fault hypothesis. In addition, it suggests which components should be replaced and isolates the reasoning in a compositional fashion.

# Chapter 3

# RTN Graph Grammar Transformations

## Contents


**3.1 Introduction** . . . . . . . . . . 48

**3.2 The Grammar for RTN-SL** . . . . . . . . . . 51

3.2.1 RTN Production Rules . . . . . . . . . . 52

**3.3 Design Transformations using Graph Grammars** . . . . . . . . . . 53

**3.4 An Example derivation using the graph grammar** . . . . . . . . . . 56

3.4.1 Scenario . . . . . . . . . . 56

3.4.2 Specification . . . . . . . . . . 57

3.4.3 Design . . . . . . . . . . 57

3.4.4 Faults . . . . . . . . . . 59

3.4.5 The chosen Template . . . . . . . . . . 60

3.4.6 Transformation . . . . . . . . . . 63

**3.5 Evaluation** . . . . . . . . . . 64


In this chapter, we outline a method for making (fault tolerant) design transformations to RTN-SL designs which provides constraints on which transformations are feasible through specifying a context. The proposal to define the abstract syntax of design notations using graph grammars is not new [Pay95]. In [Pay95], Paynter proposed a node-labelling controlled embedding (NCE) graph grammar to give semantics to an arbitrary MASCOT design. We extend that approach here, by using a context-sensitive grammar

Before we present the transformations, we first outline the idea of a graph grammar.

## 3.1 Introduction

There are many graph grammar proposals in the literature, e.g. those of [AKTY99][Roz87]. We have chosen an NCE *context-sensitive graph grammar* (NCE-CSG) [AKTY99] which is suitable to specify the abstract syntax for RTN-SL. We require a context-sensitive grammar to extend the previous work ([Pay95]) to permit the specification of fault-tolerant design templates and to restrict their application. That is, the

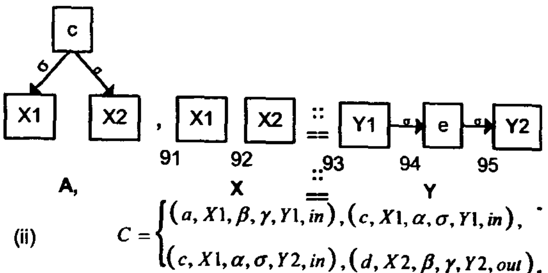



Figure 3.1: Production rule, $p$

type of templates we seek to present should not be instantiated into any part of a design, rather only an identified sub-system which is hypothesised to exhibit the faults we wish to tolerate.

We first present some basic terminology and production rules of an NCE-CSG.

**Definition 3.1 (Graph)** *Let $\Sigma$ be an alphabet of node labels and let $\Gamma$ be an alphabet of edge labels. A graph over $\Sigma$ and $\Gamma$ is a 3-tuple $D = (V, E, \lambda)$, where:*

- *$V$ is a finite nonempty set of nodes,*
- *$E \subseteq \{(v, y, w) \mid v, w \in V,\ v \neq w,\ y \in \Gamma\}$ is a set of edges,*
- *$\lambda : V \rightarrow \Sigma$ is a node labelling function*

*The components of a graph $H$ are denoted by $V_H$, $E_H$ and $\lambda$, respectively [AKTY99].*

**Definition 3.2 (Subgraph)** *Given two graphs, $H$ and $G$, we say $H$ is a subgraph of $G$ (or $H$ is induced of $G$) if $V_H \subseteq V_G$ and $\forall n \in V_H, \exists n' \in V_G \cdot abs_edge(E_H, n) = abs_edge(E_G, n')$ where $abs_edge$ : $edge_set \times node \rightarrow edge_set$ gives the set of edges whose source or target is the node in graph.*

**Definition 3.3 (A Directed Edge Neighbourhood-Controlled Embedding (edNCE) Grammar)** *The edNCE graph grammar [AKTY99] is a 6-tuple $G = (\Sigma_n, \Sigma_t, \Gamma_n, \Gamma_t, Z, P)$, where $\Sigma_n$ and $\Sigma_t$ are sets of non-terminal and terminal node labels respectively, $\Gamma_n$ and $\Gamma_t$ are sets of non-terminal and terminal edge labels respectively. $Z$ is the initial graph and $P$ is a set of production rules for transforming $Z$. A rule $p \in P$ has the form $(A, X) ::== (B, Y), C$ where $A$, $B$, $X$ and $Y$ are graphs, $X$ is a subgraph of $A$, $Y$ a subgraph of $B$, and $C$ is the embedding relation for $p$. It defines how $X$ can be replaced by $Y$ in the context of $A$ and $B$, as specified by $C$.*

Figures 3.1, 3.2 & 3.3 show an example of applying a production rule $p$, depicted in Figure 3.1, and transforming the Host graph $H$ (Figure 3.2) into the Result graph $\overleftarrow{H}$ (Figure 3.3).

The production rule of Figure 3.1 has two components: the parent/daughter graphs and the embedding relation. The parent graph in this instance has an additional context graph ($A$) component which restricts when a production is applicable. The embedding relation $C$ states that where there exists in $H$ an edge that

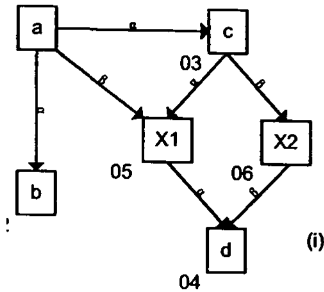



Figure 3.2: Host graph, $H$

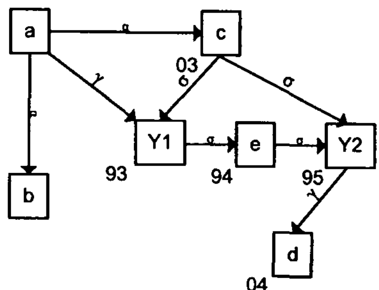



Figure 3.3: Result Graph, $\overleftarrow{H}$

is between nodes $a$ and $x1$, $x1 \in X$, and labelled $\beta$, there should exist in $\overleftarrow{H}$ an edge that is between $a$ and $y1$, $y1 \in Y$, and labelled $\gamma$; the field IN indicates the edge under consideration must be an incoming one incident on $x1$.

We observe from Figure 3.2 that there exists an arrangement of nodes equal to the mother graph ($X$) of our production rule. Additionally, the graph has the required context. We can therefore remove the mother graph from the host graph and replace with the daughter graph. The resulting graph is shown in Figure 3.3.

The graphs $X$ and $Y$ in a production rule of the form $p = ((A,X) :: == ((B,Y),C))$ are often termed the *mother* and the *daughter* graphs respectively; the context graphs $A$ or $B$ may be omitted in which case $A$ is $X$ and $B$ is $Y$. For $p$ to be applied over a host graph $H$, $A$ must be a subgraph of $H$. When all nodes and edges of $X$ are removed from $H$, the remaining graph is referred as the *rest* graph. In certain scenarios, a production of the form $p = ((A,X) :: == Y,C)$ is not sufficient to describe a context sensitive embedding. For example, the neighbourhood of a production may have surplus nodes that satisfy the embedding relation which are not required, we therefore specify an embedding context graph, $B$. A production rule $p = (A,X) :: == (B,Y),C$, where $K = A\text{-}X = B\text{-}Y$ specifies how additionally the daughter graph is embedded in the context $K$ in a production - a specification not given in $C$. An example illustrating this embedding is shown in Section 7.1.

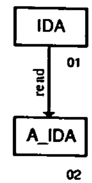



Figure 3.4: The RTN-SL Initial Graph

## 3.2 The Grammar for RTN-SL

Based on the semantics of the edNCE graph grammars, we have developed a 24-rule graph grammar which allows us to represent an RTN design in terms of a more abstract syntax. These rules can be used, in appropriate permutations, to represent any well-formed RTN design. All design representations are derived from the Initial Graph which is shown in Figure 3.4.

The 24-rule RTN grammar is a 6-tuple $G = (\Sigma_n, \Sigma_t, \Gamma_n, \Gamma_t, Z, P)$, where:

- $\Sigma_n = \{< A_IDA >, < ACT >, < IDA >, < SM >, < STATE >, < D_ST >\}$ and
- $\Sigma_t = \{< ida >, < d_st >, < s_st >, < port >\}$

are sets of non-terminal and terminal node labels respectively;

- $\Gamma_n = \{< rd_* >, < we_* >, < c1 >, < c2 >\}$

is the set of non-terminal and terminal edge labels[1]. $Z$ is the Initial Graph shown in Figure 3.4, and $P$ the set of production rules. Note that more than one node can have the same label drawn from the finite set and nodes are uniquely identified by integer tokens "01" or "02" as in Figure 3.4. The 24-rule of our grammar are grouped into 'families' based on the context in which they can be applied. The reader is referred to Appendix ?? for a complete presentation of the grammar.

### 3.2.1 RTN Production Rules

We now give a flavour of the full 24-rule of our grammar presented in graphical form. The reader is referred to [OE02] for the full presentation of the graph grammar and the abstract syntax and embedding in VDM-SL[2]. We group similar rules into 'families' dependent on their mother graphs. Then, for each rule, we describe the production rules daughter graphs. The embedding relations applicable for each rule in a 'family', are not presented here and the reader is again referred to [OE02] for a complete description.

[1] Due to the nature of our grammar –where each edge label is unique to the arc– we take *artistic* license with *read*, *write* and *boolean* labels respectively.

[2] The abstract syntax of our grammar is embedding in VDM-SL to allow for an exploration of automating the process of applying graph grammar productions and begin to specify the requirements for a tool set.

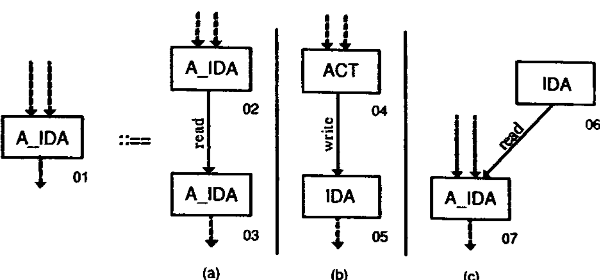



Figure 3.5: The A_IDA *family* based production rules and embedding relations

The first family of (three) rules shown in Figure 3.5 specifies how the non-terminal node labelled *A_IDA* can be transformed into networks. An *A_IDA* always consumes data supplied by activities and produces an output for an *IDA*. The first rule (a) shows how a new *A_IDA* can be instantiated into a network. The second rule (b) specifies the basic decomposition of an *A_IDA* into a pair of *ACT* and *IDA* nodes to represent an activity (*ACT*) writing to an *IDA*. The third rule (c) indicates how data from a new data source can be instantiated into an existing area of a network.

The next group of production rules, shown in Figure 3.6, are the first ***context-sensitive*** production rules to appear. These are intended to (a) direct the input from one **specific** inwardly direct ***port*** to a single state. States may only read from one port, therefore any existing inwardly directed data flows to node 01 are directed to node 03 in the production (specified in the embedding relations). Rules (b) and (c) direct output to a specific port and similarly specific input & output to a state respectively. Details to note are the repetition of the context graph on both sides of the production. The context graph $K$, where $K = X\text{-}A = Y\text{-}B$ specify how the mother ($X$) and therefore daughter ($Y$) graphs must be connected to the rest graph before and after the production respectively. However, the embedding relation $c = (\Sigma, V_X, \Gamma, \Gamma, V_Y, \{\text{IN}, \text{OUT}\})$ must not specify nodes in $V_x$ as nodes in $K$ ($V_K \not\subset V_X$), which respects the context-sensitive embedding. Should there exist a node in $V_K$ in the embedding relation then the desired finite detail is lost and the rules are not context-sensitive.

## 3.3 Design Transformations using Graph Grammars

Although the 24-rule grammar introduced previously is sufficiently rich to derive all permissible designs, our focus is on transforming existing designs rather than deriving new ones. However, the 24-rule grammar does serve to illustrate that the grammar on which the fault tolerant transformations are based is sound.

We wish to transform designs by applying templates -described in the graph grammar syntax- which represent classical fault tolerance strategies. By encoding each strategy as graph grammar production rules we harness the transformation aspect of graph grammars. Both the presentation of mother and daughter graphs and –more importantly– the embedding relation enable ‘faulty’ components to be removed from a design and replaced with an arrangement of components that tolerates the fault and preserves the *interfaces*. By ‘preserve’, it is meant that each connection from the host graph that existed previously, is replaced by a

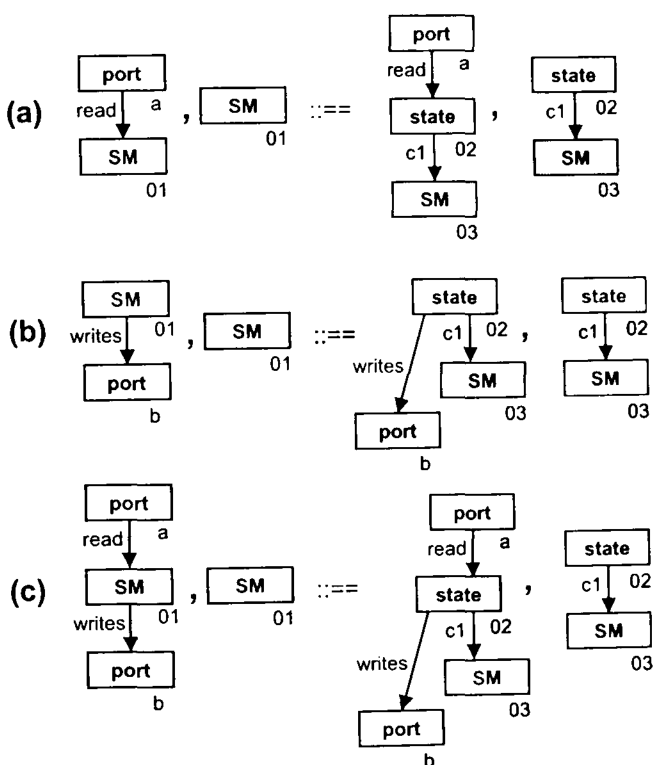



Figure 3.6: Context Sensitive SM productions and embedding relations

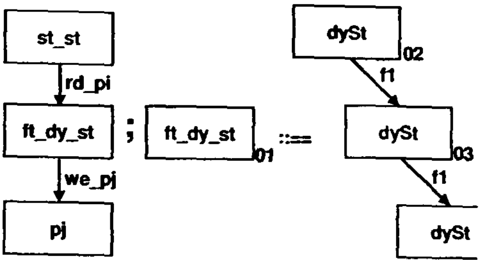



Figure 3.7: Passive state template

connection from the rest graph to daughter graph. Embedding relations inherently give us the control to describe this aspect of the transformation.

Our chosen transformational design technology is *graph grammars* for three principal reasons: (i) it is possible to represent a well-formed RTN design in an equivalent graph grammar design; (ii) production rules inherent in graph grammars allow one graph to be embedded into another (*host*) graph in a specified manner; and (iii) certain properties (e.g., flatness) desired of an RTN design can be verified effectively in the equivalent graph grammar representation. For instance, Paynter used a graph grammar to verify properties desired from a low-level design specification [Pay95].

## Context Sensitive

Suppose a context-sensitive grammar had not been used. Then, for example, the presentation of the essence of a production, such as shown in Figure 3.7 would be complicated with repeating the context of the parent graph again with the daughter graph, which also then extends the neighbourhood of the transformation further complicating the embedding relation. This undue complexity is removed by using a context sensitive grammar.

Given the previous exposition of design transformations using graph grammars, we now describe one example of the fault tolerant templates, our ultimate goal. The template shown in Figure 3.7 is termed *passive state replication* in which one faulty dynamic state (*ft_dy_st*) is replaced with a series of three similar[3] states. The context of this variation[4] of the rule is that the parent graph must write an output to some port. The embedding relation specifies that each incoming state-machine transition to 01 is replaced by a state-machine transition to 02 and that each outward transition from 01 is replaced by one from each replica. The embedding relation also creates the links that specify each replica writes to the output port.

[3]They may still exhibit the assumed fault

[4]Other possibilities are read or write only & reads and writes.

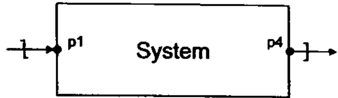



Figure 3.8: Abstract System Specification

### Advantages

By providing a transformational design method as we propose, we have established a mechanism which constrains the transformations. By choosing to use a graph grammar we retain the graphical structure inherent to RTNs, but gain a transformation mechanism for which we can describe our templates in suitable abstract form. Having chosen to use a context sensitive graph grammar, we have already argued how the transformations are more concise and benefits us by restricting the disruption on the surrounding network.

## 3.4 An Example derivation using the graph grammar

We now introduce a RTN-SL design fragment for a small example which illustrates how an RTN-SL specified real-time system can be transformed and reasoned about with respect to faults and failures. This running example will first serve to make concrete our understanding of the definitions of system and specification, then the specification of faults with respect an RTN. We then propose a simple transformation which will serve as our continuing example in subsequent chapters.

### 3.4.1 Scenario

We consider a target tracking sub-system as our example. Typical constraints on such a system are that it should produce updated vector information of the target to port, $p4$ within some time of a new image data arriving at port, $p1$, and that it should not produce a vector update if there has not been a new image.

### 3.4.2 Specification

From this scenario, we can extract **two** end-to-end specification properties. One is a timeliness and the other a safety property. One typically needs to specify both kinds of properties for real-time systems. Work has been done on showing how many of the common real-time system requirements can be formally expressed in the RTL logic [JMS88]. The timeliness and safety properties for this system are no exception.

The *timeliness* property the system should exhibit can be formalised as:

**Theorem 3.1 (Liveliness)** $\forall i : Occ, t_1 : Time \cdot \Theta(R_{p1}, i, t_1) \Rightarrow \exists t_2 : Time \cdot \Theta(W_{p4}, i, t_2) \wedge t_2 \leq t_1 + X$

This state that every input at $p1$ should give rise to an output within $X$ time units at $p4$, where $X$ is the system deadline.

A *safety* property the system should exhibit can be formalised as:

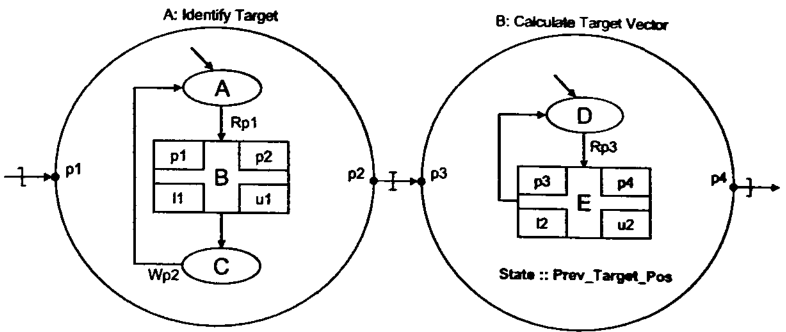



Figure 3.9: A Target Tracker

**Theorem 3.2 (Safety)** $\forall i : Occ, t_1 : Time \cdot \Theta(W_{p4}, i, t_1) \Rightarrow \exists t_2 : Time \cdot \Theta(R_{p1}, i, t_2) \wedge t_2 < t_1$

This states that every output (i.e. each occurrence) should be in response to an earlier input.

## 3.4.3 Design

The network in Figure 3.9 consists of two activities linked by a channel, and is a typical design fragment which might be found in many real-time tracking systems. The example is motivated due to [PAH00].

The arrival of fresh target images from some input sensors triggers activity A, which performs some image processing to determine the coordinates of the target. These coordinates are then passed to B via the channel, and B calculates how the target is moving, and outputs this vector to the rest of the system.

Space does not permit a full RTN-SL specification for the system to be given. Instead, the graphical specification (a subset of the textual specification) shows the arrangement of activities, IDAs, ports and state-machine representation of the functional behaviour of each activity in Figure 3.9. Where required, the textual specification is given to highlight functional behaviour which must be preserved, or highlight the specifics of the components we intend to transform.

### 3.4.3.1 RTN-SL Design Fragment

The textual fragment that follows supplements the graphical specification of Figure 3.9 to illustrate the tight coupling between the graphical form and textual specification. This aspect, though very useful at the production level, makes it tedious to pursue our objective of design transformations in a larger scale.

```
LOCAL STATE
   Prev_Target_Pos  :  T1;
END LOCAL STATE;

op assess_Threat  (image  :  Image.Processed_Image)
                                         assess  :  Threat.Threat_Assessment;
```

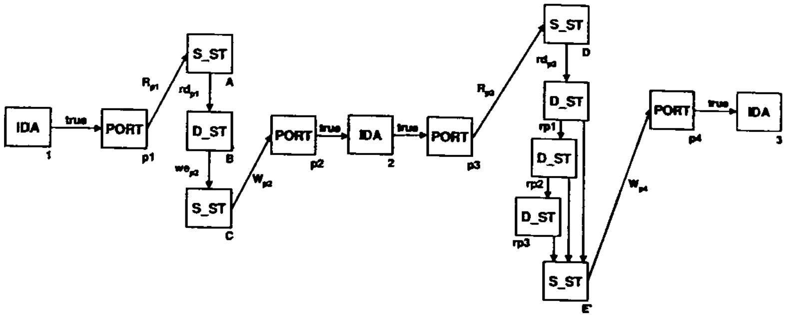


Figure 3.10: Host Graph, $G_H$

```
   EXT READ ...
   EXT WRITE ...
   PRE true;
   POST exists (tr: Threat.Threat_Assessment): assess = tr;
end op;

static stateD
   transition goes stateE on Rp3;
end state;

dynamic stateE
   op asses_threat reads from p3 writes to p4;
   [_,_,_];
   transition goes to stateD on true;
end state;
```

#### Graph Grammar representation of $G_H$

Figure 3.10 illustrates the graph grammar presentation of the design of Figure 3.9.

### 3.4.4 Faults

In the absence of a SHARD [MP98] investigation on the eventual design, we hypothesise the faults plausible in our example which could have a detrimental affect. Drawing from the system requirements, and to make this example interesting, we consider a *timing* fault within the system, specifically a *late* write fault at port $p4$.

#### Fault Hypothesis

A fault that causes a component to produce the expected value for a given nonempty input sequence too late will be termed a *late timing* fault, which in RTL can be defined as:

$$LATE_FAULT : Event \times Occ \times Interval \longrightarrow \mathbb{B}$$
$$LATE_FAULT(e,i,I) \triangleq$$
$$\exists t' : Time \cdot t' \geq I.u \wedge \Theta(e,i,t')$$

The basic RTL formula states that an event, $e$ occurs for some occurence, $i$ at some time, $t$. Th *LATE_FAULT* formula states that an event ($e$) occurs for the of the interval ($I$). This auxiliary function (and the fault axiom stated below) are discussed further in Section 4.2 when the Real-Time Logic (RTL) is introduced. The aim of the auxiliary function is to state the characteristics of the specific fault (Similar functions are defined for Early, Omit and Commit faults) so that the fault axioms can clearly identify the observable interval an event should occur within.

Specific to our example a late write at port $p4$, in RTL, is specified to identify the time the read event occured (for the same occurence) and that the write event occurs later than some deadline ($X$).

$$LateWrite(p4,i,t) \triangleq$$
$$\exists t' : Time \cdot t' \leq t \wedge \Theta(R_{p1},i,t') \wedge$$
$$LATE_FAULT(W_{p4},i,[t',t'+X])$$

In turn, this fault hypothesis would be considered for each design component, to establish where the fault in a *component*[5] leads to a fault in the *system*.

Note, extending the specification to include our fault hypothesis leads to a contradiction against the safety and timeliness properties. This contradiction can be seen by looking at the timeliness property and the fault hypothesis. The timeliness property states that the output must be produced within some deadline ($X$), whereas the fault hypothesis states that the output event will not occur until after this deadline. This contradiction must be addressed by the fault tolerant template.

### 3.4.5 The chosen Template

The RTN-SL graph grammar design fragment shown in Figure 3.12 illustrates the arrangement of design components proposed to tolerate the fault identified above. This *passive state replication* (PSR) design template requires three diverse functions[6] which can calculate a target vector.

#### Design Context

The environment of a template in a host graph is crucially important. The interactions between an environment and template may be either flows of data or control algorithms. The existing links which define the environment not only specify what new links should be established during the embedding, but also the context on which our context-sensitive grammar relies. From Figure 3.12, we can see the context (shown in Figure 3.11) specified for this example is that a dynamic state must read an input and write an output: only in this context is the template applicable.

Considering the template in Figure 3.12 more closely, the environment we consider from the host graph (Figure 3.10) on the IN direction is the arc $D \rightarrow E$ and similarly $E \rightarrow p4$ on the OUT direction. When evaluating the embedding relation it is these arcs that stipulate those required during the embedding phase.

[5] i.e. that a late exit from a dynamic state (that writes to $p4$) leads to a late write

[6] The existence, and design of such functions is assumed.

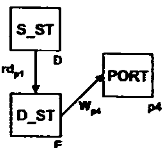



Figure 3.11: Production Rule Context

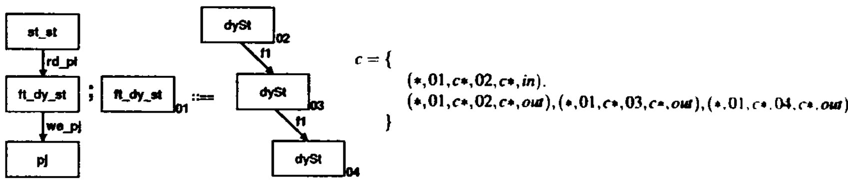



Figure 3.12: Graph Grammar Syntax of PSR Template

Specific to this context and fault tolerant pattern is the issue of a *blocking* protocol at port *p*3. The characteristics of a blocking protocol are that a reader is held up and the data is **consumed** by the reader (c.f. Section 2.4). Therefore the template must be specific in dealing with this issue and highlights why a context sensitive grammar is required. Should *rp1* fail, the data acquired by *rp1* reading *p*3 would be destroyed and not available for *rp2* or *rp3*. The solution is discussed with the production rule applicable.

### Production Rule

The rule, shown in Figure 3.12, states that, should the first dynamic state (with an associated operation) fail to produce a result within some given interval (Worst Case Response Time (WCRT)), a *failure* transition is taken to the second replica, which similarly has an upper bound to produce a result. Should this replica fail, a further *failure* transition is followed to a third and final state. The embedding relation, *c* of Figure 3.12 additionally specifies the *successful* transitions from each replica: if a replica produces the result within interval WCRT, a transition to the *successor* should occur. The fault tolerant property of this configuration is timing (and crash) tolerance.

Note, the mother of this production rule could be decorated with a context graph, one which includes the parent graph and specifies the context in which it can be replaced, however in this example no context graph is required.

### RTN-SL

Supporting specification not contained in the graph grammar presentation of this template is given below.

```
static stateD
   transition goes to stateE'_rp1 on Rp3;
end state;

dynamic stateE'_rp1
   op ___ peeps from p3 writes to p4;
   [E'_rp1_l,_,E'_rp1_u];
   transition goes to stateD on true;
   late_fault  => transition goes to stateE_rp2;
end state;

dynamic stateE'_rp2
   op ___ peeps from p3 writes to p4;
   [E'_rp2_l,_,E'_rp2_u];
   transition goes to stateD on true;
   late_fault  => transition goes to stateE_rp3;
end state;

dynamic stateE'_rp3
   op ___ reads from p3 writes to p4;
   [E'_rp3_l,_,E'_rp3_u];
   transition goes to stateD on true;
end state;
```

Considering an issue raised previously - that the data acquired by *rp1* reading $p3$ would be destroyed and not available for *rp2* or *rp3* - we specify in the supplemental RTN-SL how the template (in the context of a read-blocking protocol) overcomes this issue. The specification is now that *rp1* & *rp2* **speculatively** read from $p3$ which means $p3$ doesn't have to be stored in the local state, $\Sigma$. The third replica, *rp3* still though performs a read as it must succeed.

### 3.4.6 Transformation

The design transformation is carried out in the following manner. Figure 3.12 showed a dynamic state (D_ST) can be replaced by a specific arrangement of three replicas. Figure 3.10 showed the host graph which represents the RTN-SL design. Removing the parent graph from the host graph gives the rest graph to which we add the daughter graph. The transformation is completed by establishing the arc's specified in the embedding relation between the rest graph and the daughter graph.

The result graph net1' obtained after applying the template is given in Figure 3.13.

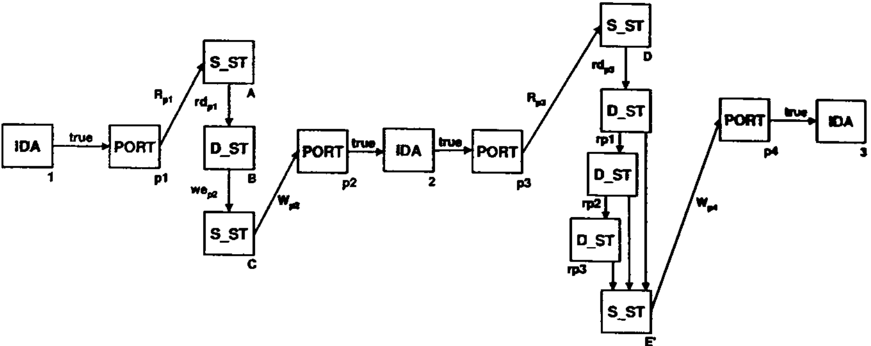



Figure 3.13: Result graph net1'

## 3.5 Evaluation

We have described a methodological way to transform an abstract representation of RTNs using graph grammars by providing templates of classical fault tolerant strategies. The method is sufficiently enabled, for example, to deal with the complexities of replication where the interface must be preserved so as to localise the environment disruption and ensure the method is compositional.

An initial investigation into defining the semantics of graph transformations, specifically those which should transform a design to a more fault tolerant one, is discussed in [OE02].

The original specification and design given above and the considered fault hypothesis are related as described in Figure 2.12. We can now begin to understand what is required to complete this example, and ultimately this thesis, to conclude i) the transformation proposed tolerates the anticipated fault, and ii) the transformed design maintains the functional properties that satisfied the original design. Building up a methodological framework from Figure 2.12, by first adding the relation shown in Figure 2.11 leads to a three-dimensional framework. The missing, or remaining, vertices to this framework are the semantic elements of our work which allows for formal verification of our designs.

It is evident we now require a semantic framework that relates each aspect of our design methodology, as now illustrated in Figure 3.14. To complete our methodology framework we are required to: define –as axioms– the semantics of each fault we wish to tolerate; show the existing axiomatic semantics sound with respect a more constructive model; and preserve the existing behaviours of RTNs under the explicit assumption that no faults occur. We therefore propose to define an operational semantic model for RTN-SL.

Each aspect of our framework is discussed, and the relationships that exist:

- *spec* :: We refer to the specification (*spec*) as end-to-end properties we wish to hold of our design, Section 3.4.2 gives two such examples.
- $FH_1$ :: Is an abstract specification of some fault behaviour. Section 3.4.4 gives an example of a *late write* fault of our abstract system specification.
- $spec \oplus FH_1$ :: This notation is illustrative to the combination, or addition, of some fault hypothesis (FH) in unison to a specification. In our example we allow for alternative behaviours, either the

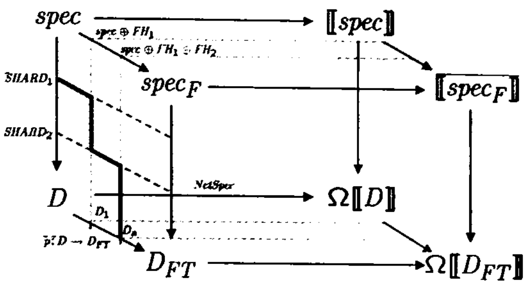



Figure 3.14: Required Semantic Framework

write event at $p4$ is timely ($\Theta(we_p4, i, t)$) or is late ($LateWrite(we_p4, i, l)$), so $spec \oplus FH_1$ can be considered as: $\Theta(we_p4, i, t) \vee LateWrite(we_p4, i, l)$. $spec_F$ is simply indicative of a specification and each FH we consider, rather than writing $spec \oplus FH_1 \oplus \ldots \oplus FH_n$.

- $D, D_{FT}$ :: A design, D is an RTN-SL representation of a solution proposed to satisfy a specification. $D_{FT}$ is indicative to the design which tolerates all faults we consider and satisfies $spec_F$. Section 3.4.3 details an example design and Section 3.4.5 shows an extract from the fault tolerant design we propose satisfies $spec_F$.

What is therefore missing is a formal framework to show the design, $D_{FT}$ does satisfy $spec_F$. In the next Part, we define more formally the specification of faults and the axiomatic semantics ($\Omega_f$) they generate. Finally, $[\![spec]\!]$ and $[\![spec_F]\!]$ are terms which express the semantic implication of our end-to-end specification in the semantic model. Although $spec$ and $FH_x$ are written as RTL predicates, $[\![spec]\!]$ and $[\![spec_F]\!]$ may appeal to other semantic properties, such as internal events, when reasoned about.

# Part II

# Semantic Descriptions

# Chapter 4

# Fault Semantics

## Contents


**4.1 Real Time Logic** . . . . . . . . 68
4.1.1 RTL Event Model . . . . . . . . 69
4.1.2 Syntax . . . . . . . . 69
**4.2 Faults in Real-Time Networks** . . . . . . . . 72
4.2.1 Fault Hypotheses . . . . . . . . 73
**4.3 Extensions to the RTN-SL language** . . . . . . . . 84
4.3.1 Language Extensions . . . . . . . . 84
4.3.2 Extended (Axiomatic) Semantics ($\Omega_f$) . . . . . . . . 90


In this Chapter, we address the issue of giving a semantics of faults, which is the core of this thesis. We first introduce the Real Time Logic (RTL) for which the existing RTN-SL semantics are axiomatically specified. Using RTL, we give an axiomatic semantics for the faults we consider in RTNs. We then introduce and describe the RTN-SL language extension and semantics we propose for describing failures.

## 4.1 Real Time Logic

Real-Time Logic (RTL) was developed for specifying and reasoning about real-time behaviours of computer-based systems. It was originally described in [JM86], since then RTL has been used for the definition of several graphical notations, specifically those for real-time systems which include Statecharts [Arm98], ADL [PAH00] and in defining the semantics of a family of communication protocols [Sim03]. RTL is used also to describe the axiomatic semantics of RTN-SL [Pay02].

### 4.1.1 RTL Event Model

The RTL event model is only informally defined in the literature [JM86] which uses a trace based model. That is, RTL reflects an early approach to reasoning about time-dependent systems. RTL makes no mention

of states or transitions, either in the language or in its model. Although an RTL interpretation can be regarded as a sequence of event sets, it is not necessary to do so, and this is different from most other techniques of modelling computer systems, which rely on transition systems. In the transition system approach, the behaviour of the system is represented by a sequence of states, which are intended to represent the system configurations. Two consequences of this distinction are important and discussed below.

First, in RTL time passes between sets of occurrences of occurring events, and the actual sets of events are instantaneous, while in transition systems, time passes in states, and transitions between them are instantaneous. Second, transition systems are generally defined by their transition, so that if two actions of the modelled system occur concurrently, a transition must be explicitly included to reflect that concurrent action, while such concurrency is implicit in RTL.

The last difference has two further results. Any computation of a transition system induces a total order on event occurrences, even those that occur *at the same time*. The fact that interleaving guarantees the existence of other computations where the ordering is different does not change this for any particular computation. Also, if two actions begin and end at the same time, transition system must introduce *faux*-states in which one action has completed and the other has not.

In order to reason formally about a specification, three things are required: a *formal language* in which assertions can be formulated; an *interpretation* of the meaning of the expressions and statements of the formal language; and a set of *axioms* and *inference rules* describing inferences which are valid for the given interpretation [BFL+94]

### 4.1.2 Syntax

We first give the formal language syntax for RTL, in which the formula's of RTL are made up of the following symbols:

- The truth symbols *true* and *false*
- A set of time variable symbols **A**
- A set of occurrence variable symbols **B**
- A set of constant symbols **C** including the natural numerals, $\mathbb{N}$
- A set of event constant symbols **D**
- The function symbol $+$
- The predicate symbols $<, \leq, >, \geq, =$
- The occurrence relation symbol $\Theta$
- The logical connectives $\wedge, \vee, \neg, \Rightarrow$
- Existential and universal quantifier symbols $\exists, \forall$

Time terms are RTL expressions built up over the constant symbols **C** and **A** which $t_1 + t_2$ is also a time term.

Occurrence terms are RTL expressions built up over the constant symbols **C** and **B** which $i+j$ is also an occurrence term

The *propositions* of RTL are constructed according to the following rules:

- The truth symbols *true* and *false* are propositions
- If $t_1$ and $t_2$ are time terms and ρ is a predicate symbol, then $t_1 \rho t_2$ is a proposition
- If $i$ and $j$ are occurrence terms and ρ is a predicate symbol, then $i \rho j$ is a proposition
- If $i$ is an occurrence term, $t$ is a time term and $e$ is an event constant, then $\Theta(e,i,t)$ is a proposition.

The *formulas* of RTL are constructed from the propositions, logical connectives and quantifiers according to the following grammar:

Key:

```
{x}      ==  x is optional
(a|b)    ==  a or b
a b      ==  a followed by b
;        ==  end of a grammar rule
:        ==  separator between non-terminals and its definition
"ABC"    ==  The terminal word 'ABC'
```

A (concrete) syntax for RTL formulas

| | | |
|---|---|---|
| formula | : | "(" formula ")" \| "¬" formula \| Bformula \| Eformula \| predicate_application \| name \| "true" \| "false"; |
| Bformula | : | formula logical_op formula \| NumbericExp comparator NumericExp; |
| Eformula | : | ("∃" \| "∀") variable_list "·" formula; |
| predicate_application | : | name "(" expression_list ")" \| th-exp; |
| logical_op | : | "∨" \| "∧" \| "⇒" \| "⇔"; |
| NumericExp | : | "(" NumericExp ")" \| "-" NumericExp \| BNumericExp \| name \| literal; |
| comparator | : | "<" \| "≤" \| ">" \| "≥" \| "=" \| "≠"; |
| variable_list | : | var_name ":" type_name {"," variable_list}; |
| expression_list | : | expression {"," expression_list}; |
| th-exp | : | "Θ" "(" event_name "," NumericExp "," NumericExp ")"; |
| BNumericExp | : | NumericExp arith_op NumericExp; |
| literal | : | number \| number "." number; |
| var_name | : | name; |
| type_name | : | name; |
| expression | : | formula \| NumericExp; |
| event_name | : | name; |
| arith_op | : | "+" \| "-" \| "*"; |

**Example 4.1 (Example RTL Formulas)** *Two typical RTL formulas that are examples of the concrete syntax are:*

$$\forall t_1 : Time, i : Occ \cdot \Theta(e_1, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(e_2.i, t_2)$$

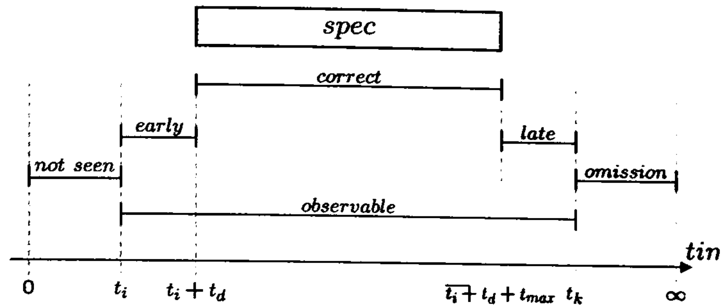



Figure 4.1: Repeat of Figure 2.5: Observable fault intervals

$$\forall t_1 : Time, i : Occ \cdot \Theta(e, i+1, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 < t_1 \wedge \Theta(e, i, t_2)$$

*The first expresses two events ($e_1$ and $e_2$) are* causally *linked, the second formula states that a subsequent occurrence of an event must occur strictly later than its predecessor.*

## 4.2 Faults in Real-Time Networks

From our informal description on fault specifications in Section 2.1.3, we now seek a more formal presentation. Using RTL, we first formalise four key definitions as auxiliary functions, then for each RTN *component* we consider the faults that can be specified on it and which will be considered in this thesis. This set of fault definitions form our fault hypotheses for RTNs. The auxiliary functions for LATE, EARLY, OMIT and COMMIT faults which are used in the RTN definition of faults are given below. The intervals highlighted in Figure 4.1 (which is a repeat of Figure 2.5) illustrated the bounds between which a fault is observable. These functions simply define whether an event is observable, or not, given the type of fault. The bounds to each interval are dependent on the fault type, Table 4.1 enumerates each bound with respect to the time markings in Figure 4.1.

An interval, $I$ is a pair of *Time* values, namely the lower ($l$) and upper ($u$) time bounds.

$$\begin{aligned} &Interval \;::\; l : Time \\ &\qquad\qquad\;\;\; u : Time \\ &\text{inv } mk\text{-}interval(l, u) \triangleq l \leq u \end{aligned}$$

$$\begin{aligned} &LATE : Event \times Occ \times Interval \rightarrow \mathbb{B} \\ &LATE(e, i, I) \triangleq \\ &\quad \exists t' : Time \cdot t' \geq I.u \wedge \Theta(e, i, t') \end{aligned}$$

$$\begin{aligned} &EARLY : Event \times Occ \times Interval \rightarrow \mathbb{B} \\ &EARLY(e, i, I) \triangleq \\ &\quad \exists t' : Time \cdot t' \leq I.l \wedge \Theta(e, i, t') \end{aligned}$$

$$OMIT : Event \times Occ \times Interval \rightarrow \mathbb{B}$$
$$OMIT(e,i,I) \triangleq$$
$$\nexists t' : Time \cdot I.l \leq t' \leq I.u \wedge \Theta(e,i,t')$$

$$COMMIT : Event \times Occ \times Interval \rightarrow \mathbb{B}$$
$$COMMIT(e,i,I) \triangleq$$
$$\exists t' : Time \cdot I.l \leq t' \leq I.u \wedge \Theta(e,i,t')$$

| Fault Type | Interval |
|---|---|
| LATE | $t_i + t_d \rightarrow t_i + t_d + t_{max}$ |
| EARLY | $t_i + t_d \rightarrow t_i + t_d + t_{max}$ |
| COMMIT | $0 \rightarrow t_i$ |
| OMIT | $t_i + t_d \rightarrow \infty$ |

Table 4.1: Bounds to Fault intervals

Note, a LATE & EARLY fault occurs after or before some interval respectively, whereas COMMIT & OMIT faults are within some interval. It should be noted, that the value $t_k$ shown in Figure 4.1 is not realised in these definitions. However, the dynamic semantics for RTNs do distinguish between *late & omit* faults and such a value is retrieved from the semantics, as is stated for each omission fault definition that follows.

### 4.2.1 Fault Hypotheses

Following on from the abstract definition of faults given previously, we now seek to apply those definitions to a specific real-time architecture, namely that of Real-Time Networks (RTNs). Identified in the definitions are bounds to the time intervals in which each fault is observable. Fortunately, such time bounds are implicit in the semantics of RTNs. For each fault identified in Table 4.2, we provide a suitable RTL proposition which specifies the dynamic behaviour over the component identified. The collection of these predicates form our *fault hypotheses* (FH) for RTNs.

Consider a RTN-SL component: a *dynamic state*. Graphically this is described in Figure 4.2, which specifies that once the state has been entered, it should be left within $u1$ time units. A *fault hypothesis* may be that the write action to the output port ($pX$) is delayed. The subsequent *fault definition* may be:

$$LateWrite(we_pX,i,t) \triangleq$$
$$\exists t' : Time \cdot t' + u1 < t \wedge \Theta(\uparrow A,i,t') \wedge$$
$$LATE(we_pX,i,[t'+l1,t'+u1])$$

This states that a *late write* fault is observed at time, $t$ which is defined with respect the time ($t'$) when state $A$ was entered and its specified time bounds i.e. the write event (which triggers the exit transition) should not occur before $l1$ time units ($t' + l1$) nor later than $u1$ time units ($t' + u1$). The auxiliary function of *LATE* distinguishes the fault from that of an omission stating the event does occur, despite being late.

If all the components of a RTN are to be considered faulty, the design transformation to a tolerant RTN would be intractable. Rather, we propose that only a sub-part of an RTN be considered faulty (identified

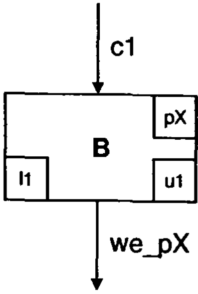



Figure 4.2: A Dynamic State

by some suitable safety assessment approach, e.g. SHARD) at a time. This requires that the extended semantics of $\Omega_f$ should specify which component is perceived as being potentially faulty. This requirement is critical and influences the RTN-SL language extension, semantics and approach to defining our fault hypothesis accordingly.

> **Definition 4.1** *A component is defined to be a* distinguishable semantic object which has discernible dynamic behaviour. *Particularly, a RTN component is one of –determined by the semantics– a static or dynamic state, Activity, Port or an IDA. A* fault *component is defined to be a component which exhibits extraordinary behaviour (as defined below) which is at conflict with the expected behaviour.*

For each fault classification, described in Chapter 2, we consider it plausibility on each RTN component.

A complete presentation of every permutation between each fault classification (i.e. late, early, value, etc) and all RTN events (i.e. state entry, read, read vale, etc) would generate over 100 fault definitions. Many of these definitions would overlap, for example, a late exit state fault would incur a late entry fault at a successor state. Instead, we identify in Table 4.2 those faults which are most meaningful and do not overlap. Those cells in Table 4.2 which are blank indicate a meaningful fault does not exist for the permutation, the shaded cells indicate a fault definition is feasible, though overlaps another.

| Events | Faults | Dynamic States | Static States | Activity |
|---|---|---|---|---|
| $\uparrow$s | early | | | |
| | late | | | |
| | value | | | |
| | omit | | | |
| | commit | | | |
| $\downarrow$s | early | *EarlyDyExit* | *EarlyStTimeExit* & *EarlyStEvExit* | |
| | late | *LateDyExit* | *LateStTimeExit* & *LateStEvExit* | |
| | value | | | |
| | omit | | | |
| | commit | | | |
| rds | early | *EarlyRead* | | |
| | late | *LateRead* | | *LateActRead* |
| | value | *ReadValue* | | |
| | omit | *OmitRead* | | |
| | commit | | | |
| wds | early | *EarlyWrite* | | |
| | late | *LateWrite* | | |
| | value | *WriteValue* | | |
| | omit | *OmitWrite* | | |
| | commit | | | |
| progress | Crash | *Crash* | | |

Table 4.2: Faults considered

The fault classifications distinguished in Table 4.2 are taken from the SHARD analysis method and those *guide words* derived in Section 4.2.

The form of the fault hypothesis definitions given below follows closely from that used to define the semantics of RTN-SL [Pay02]. Each FH has the form: *every syntactic entity in the state-machine, which satisfies certain static connection properties, will exhibit some specified dynamic behaviour.* However, we do not intend to consider all consituent components of a RTN to be faulty. Instead, we only consider those components to be faulty which are found by the SHARD analysis.

As an aid to presentation, the quantification over the syntactic entities of the RTN-SL specification is left implicit, and the static syntactic guards are presented in square brackets at the start of each schema. To support our specifications below, we require several auxiliary functions which determine the event name associated with components under certain actions, such as read and write actions at ports and entry and exit actions of states, as illustrated below:

$rds : Port \rightarrow Event$

$wds : Port \rightarrow Event$

$\uparrow : State \rightarrow Event$

As stated previously, the bounds $t_i$, $t_j$, $t_d$, $\Delta t$ and $t_{max}$ from Figure 4.1 are realised within the RTN architecture either by time bounds on events, deadlines on components or the occurrence of other events, e.g. Omission: $\infty$ could be defined as re-entering the state for the $i+1$ occurrence. These bounds can be deter-

mined in the context of RTN components and are defined and discussed in the following RTL definitions of faults. For each definition, a brief commentary will describe the expected dynamic behaviour which determines each fault **in isolation**.

#### 4.2.1.1 State Machine Component Faults

The constituent components of state machine are its (static and dynamic) state which we consider as atomic. Therefore, all the behaviours of an RTN can be defined as state-machine actions:

*LateRead* :: **Dynamic State, Late Read Fault –** Given a dynamic state performs a read for the *ith* occurrence, then a *late read fault* occurs should the read not start immediately upon entering the associated dynamic state.

*LATE* specifies an event occurs after the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\Uparrow s$ | A read cannot start before entering the associated (non-faulty) state. |
| upper | $s.wcrt$ | If a state is supposed to read and hasn't before the state's WCRT deadline, then the *read* is *late* |

$$[is_DynamicState(s), s.in_p \neq nil, inHolding(s.in_p)]$$
$$LateRead(s.in_p, i, t) \triangleq$$
$$\exists t', t'' : Time \cdot t' < t \leq t'' \wedge \Theta(\Uparrow s, i, t') \wedge \neg\Theta(\Downarrow s, i, t'') \wedge$$
$$LATE(rds(s.in_p), i, [t', t' + s.wcrt])$$

Note, this definition weakens the existing semantics defined by $\Omega$, which states that a read **must** occur at the same time as entering the (dynamic) state. By the laws of RTL, $t$ is bound to occur later than $I.u$ given the definition of LATE.

*LateWrite* :: **Dynamic State, Late Write Fault –** Similarly, should a dynamic state be specified to perform a write, then a *late write fault* occurs should the write commence later than the exit transition from the associated dynamic state:

*LATE* specifies an event occurs after the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\Uparrow s + bcet$ | A write cannot commence before the *best case* deadline. |
| upper | $\Uparrow s + wcet$ | A write must occur before the *worst case* deadline. |

$$[is_DynamicState(s), NonHolding(s.out_p), \neg is_Speculative(s.out_p)]$$
$$LateWrite(s.out_p, i, t) \triangleq$$
$$\exists t', t'' : Time \cdot t' < t \leq t'' \wedge \Theta(\Uparrow s, i, t') \wedge \neg\Theta(\Downarrow s, i, t'') \wedge$$
$$LATE(wds(s.out_p), i, [t' + s.bcet, t' + s.wcet])$$

*OmitRead* :: **Dynamic State, Read Omission Fault –** Eventually a *late read fault* becomes an omission fault. However, unlike the definitions in Section 2.1.3, we can enumerate $\infty$ to a more precise, discernible value:

*OMIT* specifies that an event does not occur within the interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\upharpoonleft s = t'$ | A read cannot start before entering the associated state. |
| upper | $t''$ | $t''$ is specified as either the time at which $s$ is exited[1], or the time at which the (holding) port is next read from (which may potentially be another state in the activity). |

$$
\begin{array}{l}
[is_DynamicState(s), InHolding(s.in_p)] \\
OmitRead(s.in_p, i, t) \triangleq \\
\quad \exists t', t'' : Time \cdot t' < t \leq t'' \wedge \Theta(\upharpoonleft s, i, t') \wedge \\
\qquad\qquad (\Theta(\downharpoonleft s, i+1, t'') \vee \Theta(rds(s.in_p), i+1, t'') \wedge \\
\qquad\qquad OMIT(rd(s.in_p), i, [t', t''])
\end{array}
$$

*OmitWrite* :: **Dynamic State, Write Omission Fault –** Similarly to *read omit fault*, the specification here makes a precise value for $\infty$ which is the upper bound of a *write omit fault:*

*OMIT* specifies that an event does not occur within the interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\downharpoonleft s = t'$ | A write cannot commence before leaving its associated state, earlier would be a *early write fault.* |
| upper | $t''$ | A write must occur before the port is next written to (otherwise an omission faults occurs) or before the dynamic state is next entered |

$$
\begin{array}{l}
[is_DynamicState(s), OutHolding(s.out_p)] \\
OmitWrite(s.out_p, i, t) \triangleq \\
\quad \exists t', t'' : Time \cdot t' < t \leq t'' \wedge \Theta(\downharpoonleft s, i, t') \wedge \\
\qquad\qquad (\Theta(\upharpoonleft s, i+1, t'') \vee \Theta(wds(s.out_p), i+1, t'') \wedge \\
\qquad\qquad OMIT(we(s.out_p), i, [t', t''])
\end{array}
$$

*EarlyRead* :: **Dynamic State, Early Read Fault –** Should an dynamic state perform a read action before the associated dynamic state is entered, then a *early read fault* has occurred:

*EARLY* specifies an event occurs before the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\upharpoonleft s$ | A read should occur when entering a (non-faulty) state |
| upper | $\upharpoonleft s + bcet$ | A read must occur before the earliest exit can occur and therefore respect its time bounds |

$$
\begin{array}{l}
[is_DynamicState(s), s.in_p \neq nil] \\
EarlyRead(s.in_p, i, t) \triangleq \\
\quad \exists t' : Time \cdot t < t' \wedge \Theta(\upharpoonleft s, i, t') \wedge \\
\qquad\qquad EARLY(rds(s.in_p), i, [t', t' + bcet])
\end{array}
$$

*EarlyWrite* :: **Dynamic State, Early Write Fault –** An *early write fault* occurs should the write occur before the best case execution time (BCET) delay:

*EARLY* specifies an event occurs before the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\uparrow s + bcet$ | A write should not start before entering the associate state and the minimum time delay has elapsed |
| upper | $\uparrow s + wcet$ | A write should occur upon leaving a state and therefore within its time bounds |

$$
\begin{array}{l}
[is_DynamicState(s), s.out_p \neq nil] \\
EarlyWrite(s.out_p, i, t) \triangleq \\
\quad \exists t' : Time \cdot t' \leq t \wedge \Theta(\uparrow s, i, t') \wedge \\
\qquad\qquad EARLY(we(s.out_p), i, [t' + bcet, t' + wcet])
\end{array}
$$

*WriteValue* :: **Dynamic State, Write Value Fault –** A *write value fault* occurs if the value written at the time of leave the associated state does not satisfy the post-condition:

$$
\begin{array}{l}
[is_DynamicState(s)] \\
WriteValue(s.out_p, i, t) \triangleq \\
\quad \exists t' : Time \cdot t' \leq t \wedge \Theta(\uparrow s, i, t') \wedge \Theta(\downarrow s, i, t) \wedge \\
\qquad\qquad pre_(s.in_p(t'), \overleftarrow{v}(t')) \wedge \\
\qquad\qquad \neg post_(s.in_p(t'), \overleftarrow{v}(t'), v(t), s.out_p(t))
\end{array}
$$

Note, the timing of $\downarrow s$ is the same as timing of the fault, which excludes timing (and therefore omission) from the definition.

*ReadValue* :: **Dynamic State, Read Value Fault –** A read *value fault* occurs if the value read at the time of entering the associated state does not satisfy the pre/post-condition:

$$
\begin{array}{l}
[is_DynamicState(s)] \\
ReadValue(s.in_p, i, t) \triangleq \\
\quad \exists t' : Time \cdot t < t' \wedge \Theta(\uparrow s, i, t) \wedge \Theta(\downarrow s, i, t') \wedge \\
\qquad\qquad pre_(s.in_p(t), \overleftarrow{v}(t)) \wedge \\
\qquad\qquad \neg post_(s.in_p(t), \overleftarrow{v}(t), v(t'), s.out_p(t'))
\end{array}
$$

*Crash* :: **Dynamic State, Crash Fault –** A *crash* fault occurs of a state, s should there have occurred an event ($\uparrow s$) entering the state and no occurrence of leaving the state ($\downarrow s$) during the interval from entering $s$ to $t$.:

*OMIT* specifies that an event does not occur within the interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\uparrow s + bcet$ | Once a (non-faulty) state is entered, it may exit any time after the minimum delay. |
| upper | $t$ | A *crash fault* is a special case of *omit*, in that the omission is bounded to $\infty$, or in our case the time at which we observe the fault, $t$. |

$$\begin{array}{l} [is_DynamicState(s), \neg is_TerminateState(s)] \\ Crash(s,i,t) \triangleq \\ \quad \exists t' : Time \cdot t' < t \wedge \Theta(\upharpoonleft s, i, t') \wedge \\ \qquad\qquad OMIT(\downharpoonleft s, i, [t' + s.bcet, t]) \wedge OMIT(\upharpoonleft s, i+1, [t' + s.bcet, t]) \end{array}$$

It is interesting to note that $I.u = t$. If a *crash* fault is observable at $t$, then it has been observable since $t'$ (see Figure 4.1). Whereas a *crash* fault (a special case of an *omit* fault) is bounded to $\infty$ in Section 2.1.3, no assertion to $I.u$ can be made greater than $t$.

*LateDyExit* :: **Dynamic State, Late Exit Fault –** Upon entering a state, the state must be exited before the deadline, otherwise a *late exit fault* has occurred:

*LATE* specifies an event occurs after the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\upharpoonleft s + bcet$ | A state cannot be exited until at least after the minimum time delay after entering the state |
| upper | $\upharpoonleft s + wcet$ | A state must exit within its deadlines |

$$\begin{array}{l} [is_DynamicState(s)] \\ LateDyExit(s,i,t) \triangleq \\ \quad \exists t' : Time \cdot t' \leq t \wedge \Theta(\upharpoonleft s, i, t') \wedge \\ \qquad\qquad LATE(\downharpoonleft s, i, [t' + s.bcet, t' + s.wcet]) \end{array}$$

*EarlyDyExit* :: **Dynamic State, Early Exit Fault –** A *early exit fault* occurs should a state exit before the minimum time delay has elapsed:

*EARLY* specifies an event occurs before the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\upharpoonleft s + bcet$ | A state cannot be exited until at least after the minimum time delay after entering the state |
| upper | $\upharpoonleft s + wcet$ | A state must exit within its deadlines |

$$\begin{array}{l} [is_DynamicState(s)] \\ EarlyDyExit(s,i,t) \triangleq \\ \quad \exists t' : Time \cdot t' \leq t \wedge \Theta(\upharpoonleft s, i, t') \wedge \\ \qquad\qquad EARLY(\downharpoonleft s, i, [t' + s.bcet, t' + s.wcet]) \end{array}$$

*EarlyStTimeExit* :: **Static State, Early (Timing) Exit Fault –** Should a static state exit before the specified time, then a *early exit fault* is said to have occurred:

*EARLY* specifies an event occurs before the correct interval, therefore the bounds here are:*EARLY*

| Interval | Bound | Justification |
|---|---|---|
| lower | $\upharpoonleft s + l$ | A static state should not be exited before the lower bound, $l$ specified. |
| upper | $\upharpoonleft s + u$ | A static state should not exit later than the upper bound, $u$ specified. |

$[is_StaticState(s), tr \in TransitionsOut(s,ts), is_TimingBounds(tr.label), tr.label.x = (l,u)]$
$EarlyStTimeExit(s,i,t) \triangleq$
$\exists t' : Time \cdot t' < t \wedge \Theta(\uparrow s, i, t') \wedge$
$EARLY(\downarrow s, i, [t'+l, t'+u])$

*EarlyStEvExit'* :: **Static State, Early (Event) Exit Fault –** Should a static state exit before the specified exit event, then a *early exit fault* is said to have occurred.

*EARLY* specifies an event occurs before the correct interval, therefore the bounds here are:*EARLY*

| Interval | Bound | Justification |
|---|---|---|
| lower | $\uparrow s = t'$ | A static state should not exit before it is entered |
| upper | $t''$ | A static state should not exit before the event label occurs. |

$[is_StaticState(s), tr \in TransitionsOut(s,ts), is_EventBounds(tr.label), tr.label.x = e]$
$EarlyStEvExit(s,i,t) \triangleq$
$\exists j : Occ, t', t'' : Time \cdot t' < t < t'' \wedge \Theta(\uparrow s, i, t') \wedge \Theta(e, j, t'') \wedge$
$EARLY(\downarrow s, i, [t', t''])$

*LateStTimeExit* :: **Static State, Late (Timing) Exit Fault –** Should a static state exit later than the specified time, then a *late exit fault* is said to have occurred.

*LATE* specifies an event occurs after the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\uparrow s + l$ | A static state should not be exited before the lower bound, $l$ specified. |
| upper | $\uparrow s + u$ | A static state should not exit later than the upper bound, $u$ specified. |

$[is_StaticState(s), tr \in TransitionsOut(s,ts), is_TimingBounds(tr.label), tr.label.x = (l,u)]$
$LateStTimeExit(s,i,t) \triangleq$
$\exists t' : Time \cdot t' + u < t \wedge \Theta(\uparrow s, i, t') \wedge$
$LATE(\downarrow s, i, [t'+l, t'+u])$

*LateStEvExit'* :: **Static State, Late (Event) Exit Fault –** Should a static state exit later than the specified exit event, then a *late exit fault* is said to have occurred.

*LATE* specifies an event occurs after the correct interval, therefore the bounds here are:

| Interval | Bound | Justification |
|---|---|---|
| lower | $\uparrow s = t'$ | A static state should not exit before it is entered |
| upper | $t''$ | A static state should not exit before the event label occurs. |

$[is_StaticState(s), tr \in TransitionsOut(s,ts), is_EventBounds(tr.label), tr.label.x = e]$
$LateStEvExit(s,i,t) \triangleq$
$\exists j : Occ, t', t'' : Time \cdot t' < t'' < t \wedge \Theta(\uparrow s, i, t_1) \wedge \Theta(e, j, t'') \wedge$
$LATE(\downarrow s, i, [t', t''])$

#### 4.2.1.2 Activity Component Faults

An activity is simply a collection of its states, transitions and ports. The fault axioms for activities define a failure at the acitivty level to behaviours of the activity's component parts. We give as an example of this approach a single definition below which is intentionally loose such that no constraint is made that state, s reads from some port.

*LateActRead* :: **Activity, Late Read Fault –**

$[is_Activity(act)]$

$LateActRead(act, i, t) \triangleq$

$\exists s \in act.SM.states \cdot late_rd(s.in_p, i, t)$

#### 4.2.1.3 IDA Component Faults

Again, our approach to specifying faults for IDAs is to regard each components as a *black box* and specify its fault definition in terms of our state-machine faults. However, this restricts the expressiveness towards an IDAs dynamic behaviour, such that no reference can be made to the internal value of an IDA. That is, only the externally visible events and their value can be used to express the fault hypothesis of these components. With regards *both sides* of an IDA, that is the reader and writer our approach is similar to that of Activities & state-machines, in that the causal behaviour is specified. Similarly, we specify the behaviour of IDAs by relating the occurrence of faults at the IDA interfaces (*ports*) to the caused occurrence of faults at either the opposite interface at the IDA or the related Activity fault. Our decision to explicitly link the cause of an IDA fault to the (faulty) dynamic behaviour of an activity (and therefore a state-machine state) is derived from the *link axioms* of RTN-SL which state the coupling of activity and IDA events. It is therefore seen to be consistent with our current approach to reasoning when one must *trace* events through an RTN. It could be argued that we do not specify fault IDA axioms, just their *link* counterparts. However, because we do not consider the internal behaviours of IDAs, we are restricted to linking the behaviours of activities by means of events. Although, the approach developed in this thesis could be applied to the full reasoning of IDA faults, we do not wish to use arrangements of them at an RTN level.

**Axiom *Ax_IDA_rd_late* :: IDA –** As an example, we illustrate the coupling by defining a *late read* fault of an IDA:

$$LateRead(ida.out, i, t) \triangleq$$

$$(OutHolding(ida.out) \Rightarrow \exists t' : Time \cdot t' < t \land \Theta(late_we(ida.in), i, t'))$$

$$\lor$$

$$(NotOutHolding(ida.out) \Rightarrow \exists t' : Time \cdot t' > t \land \Theta(value_rd(ida.out), i, t))$$

## 4.3 Extensions to the RTN-SL language

The current version of RTN-SL, version 3.1 and its semantics are presented in [Pay02]. There, the concrete and abstract syntaxes are presented along with the static and dynamic semantics, the latter as RTL axioms.

The dynamic semantics are presented as a function, $\Omega$ which generates the RTL axioms from a RTN-SL specification which specify the behaviours of the constituent components. The set of axioms constitute a PVS theory, which is motivated by the intention to reason about RTN-SL specifications using the PVS theorem prover. In this section we propose the additions and modifications to this language to support the specification of faults in RTN-SL. We refer to the extended (dynamic) semantics as the function, $\Omega_f$.

### 4.3.1 Language Extensions

Investigations and experience from the initial stages of our research lead to the requirement of several new RTN-SL language extensions. This section presents the concrete grammar, abstract syntax, static and dynamic semantics necessary to include our requirements into the RTN-SL language. Where appropriate, we highlight our modifications to the dynamic semantics presented in $\Omega$, for each extension outlined. The axiomatic schemata of $\Omega$, is present as 13 axioms referred by their number ($Ax1,\ldots,Ax13$). We present all the extensions to $\Omega$ together in Section 4.3.2 and not with each language extension, though we refer the reader to the appropriate axiom modifications generated by each extension. We therefore continue to use the same names as references in these modifications.

#### 4.3.1.1 Multiple Static State Exit Transitions

The motivation to limit the number of exit transitions from a static state to one was simplicity, then we propose to lift this restriction. Our motivation is to permit fault transitions from static state, as identified in Table 4.2. It is only the static semantics that need be modified, though with a slight influence on the dynamic semantics that necessitates the conjunction of exit guards on each transition.

#### 4.3.1.2 Fault Transitions

Experience in attempts to write RTN specifications has highlighted the importance of state transitions as the predominant RTN component for specifying fault actions. A *fault action* is just another kind of operation in an RTN that are performed at random time intervals on a RTN by a network's adverse environment. This allows one to describe fault actions by definitions similar to those used to describe the semantics of ordinary operations. It is therefore preferred that the semantics of faults should be mapped to transitions. An example of the suggested concrete syntax is given below (Specification 4.1). Transitions labelled without a fault hypothesis, FH assume no faults can occur, this approach is similar the approach taken for speculative read and writes.

The required extensions to the specification of RTN-SL are outlined below.

**Concrete Grammar**

```
fault_transition : "LATE_FAULT" | "EARLY_FAULT" | "VALUE_FAULT" |
                   "OMISSION_FAULT";
```

**Specification 4.1** Example RTN-SL Fault Transition Concrete Syntax

```
dynamic state_A
  op foo;
  timing [bcet, wcrt, wcet]
  transition goes to state_B on true;
  late_fault => transition goes to state_C';
  value_fault => transition goes to state_D';
end state;
```

```
transition_guard : "WRITE" "FAILURE" "=>" | "WRITE" "SUCCESS" "=>" |
                   fault_transition "=>" ;

dynamic_transition_def : {transition_guard} "TRANSITION" "GOES" "TO"
                                                 name "ON" expression ";"

dynamic_transition_defs : dynamic_transition_def {dynamic_transition_def} ;

dynamic_state : "DYNAMIC"
                   name
                   local_op_defs
                   {timing_def}
                   dynamic_transition_defs
                "END" "STATE" ;
```

**Abstract Syntax**

```
Transition : TYPE = { tr : [# Guard : Transition_Guard, %-- optional?
                              Source : State,
                              Target : State,
                              Label : Label_Type #] |
                 inv_Transition(Guard(tr), Source(tr), Target(tr), Label(tr)) }
```

#### 4.3.1.3 *Speculative* Reads

It was realised that the existing read and write access methods were too strict to adequately specify fault tolerant strategies. The methods forced complicated algorithms for reading and recording values to an activities local states which are then subsequently used withing calculations. It was preferred to diversify these methods and complement them with *speculative* read and write methods. This meant a reader could *peep* read a value from a blocking protocol which meant the read wasn't destructive nor blocking. Similarly, a speculative write meant a writer couldn't be help up. Although the implementation of speculative reads

are completed in RTN-SL v3.1, the documentation in [Pay02] lacks the fuller explanation and definitions given below.

**Specification 4.2** Example RTN-SL spec/peep concrete syntax

```
ports
  p1 : (channel, integer, in, speculative);
  p2 : (channel, integer, out, speculative);
end ports

dynamic state_A
  op foo peeps from p1 writes to p2;
  timing [bcet, wcrt, wcet]
  transition goes to state_B on true;
  read_failure => transition goes to state_C' on true;
  write_success => transition goes to state_B on true;
  write_failure => transition goes to state_C' on true
end state;
```

Given transition guards are optional, their absence assumes the default behaviour of success. Specifically to speculative reads this means a read or write action succeeds, for example, the first and third transition specification in Specification 4.2 are identical. If they had each specified alternative target states, then one of the transition would be chosen non-deterministically.

The required extensions to the specification of RTN-SL are outlined below.

### Abstract Syntax & Static Semantics

```
access_type : TYPE = {normal, speculative}

read_type : TYPE = {read, peep}

inv_Port(ident : Id_Name,
         type : access_type,
         direction : Direction_Kind,
         protocol : Protocol_kind,
         the_dataType : A_type_reference) : bool =
     (type = speculative IFF (InHolding(protocol) OR OutHolding(protocol)) AND
                              NOT Stimming(protocol))

Port : Type = { port : [# ident : Id_Name,
                          type : access_type,
                          direction : Direction_Kind,
                          protocol : Protocol_kind,
                          the_dataType : A_type_reference #] |
inv_Port(ident(port), type(port), direction(port),
```

```
        protocol(port), the_dataType(port))

DestructiveRead (p : Port) : bool =
        type(p) = speculative

Speculative (p : Port) : bool =
        type(p) = normal

inv_IDA_Spec(Ida_id : Id_Name,
            type : access_type,
            withed_adt : optional_type[Id_Name],
            Kind : Protocol_Kind,
            Datatype_ref : A_type_reference,
            Buffersize : optional_type[nat]): bool =
    (Kind = Poll IFF absent?(Buffersize)) AND
    (Kind = Dataless_Channel OR Kind = Stimulus) IFF
            (basic_type?(Datatype_ref) AND type_name(Datatype_ref)=NULL_TYPE)) AND
    (type = speculative IFF (InHolding(Kind) OR OutHolding(Kind)) AND
                            NOT Stimming(Kind))

IDA_Spec : Type = { ida : [# Ida_id : Id_Name,
                            type : access_type,
                            withed_adt : optional_type[Id_Name],
                            Kind : Protocol_Kind,
                            Datatype_ref : A_type_reference,
                            Buffersize : optional_type[nat] #] |
inv_IDA_Spec(Ida_id(ida), withed_adt(ida), Kind(ida),
            Datetype_ref(ida), Buffersize(ida))}

inv_Operation_localisation( op : Operation,
                            in_p : operational_type[Port],
                            in_mode : read_type
                            in_wcrt : optional_type[Time],
                            out_p : optional_type[Port] ) : bool =
    (absent?(in_p) AND absent?(input_parameter(op)) AND absent?(in_wcrt)) OR
    (present?(in_p) AND direction(x(in_p)) = in_dir AND
     present?(input_parameter(op)) AND NOT Stimming(x(in_p))) AND
     ((absent?(out_p) AND absent?(output_result(op))) OR
     (present?(out_p) AND direction(x(out_p)) = out_dir AND
      present?(output_result(op))) AND
     (present?(in_wcrt) IMPLIES present?(in_p))) AND
    (in_mode = peep IFF speculative(in_p)) AND
    (in_mode = peep IFF DestructiveRead(type(in_p)) )
```

```
Operation_Localisation : TYPE = (l_op : [# Op : Operation,
                                           Input_Port : operational_type[Port],
                                           Input_Read_Mode : read_type
                                           Input_WCRT : optional_type[Time],
                                           Output_Port : optional_type[Port] #] |
inv_Operation_Localisation(Op(l_op), Input_Port(l_op), Input_Read_Mode(l_op),
                           Input_WCRT(l_op), Output_Port(l_op)) }

at_least_one_read_on_blocking_port( in_ps : finite_set[Port],
                                    ss : finite_set[State] ) : bool
     FORALL (p : {x : in_ps | blocking?(x)} ) :
     ( EXISTS(s : {x : State | member(x, ss) AND dynamic_state?(s)} ) :
        ( EXISTS (op_l : ops(s)) :
           (Input_Port(op_l) = p IMPLIES Input_Read_mode(op_l) = read)
        )
     )

inv_activity_Spec ( a_id : Id_Name,
                    adts : finite_set[Id_Name],
                    ts : finite_set[A-Type_Declaration],
                    in_ps, out_ps : finite_set[Port],
                    aux_defs : finite_sequence[Auxiliary_Definition],
                    ls : finite_set[Variable],
                    ops : finite_set[Operation],
                    sm : SM_Spec ) : bool =
    FORALL (v1,v2 : {x : Variable | member(x,ls) }):
         (var_id(v1) = var_id(v2) IMPLIES v1 = v2) AND
    FORALL (s : {x : State | member(x, States(sm)) }):
         Only_valid_ports_accessed_by_states(s,union(in_ps,out_ps)) AND
    FORALL (t : {x : Transition | member(x,Transitions(sm)) }):
         Ports_on_events_are_valid(t,union(in_ps,out_ps)) AND
    at_least_one_read_on_blocking_port(in_ps, States(SM_Spec))
```

### 4.3.2 Extended (Axiomatic) Semantics ($\Omega_f$)

This section details the extended axiomatic semantics to $\Omega$ given in [Pay02] which considers the behaviour of faults. This extension describes the semantic function, $\Omega_f$.

As was the approach in defining $\Omega$ (in [Pay02]), we again present the schema definitions for $\Omega_f$. This approach specifies the complete behaviours of a fault RTN component. The remaining issue is to distinguish when we apply each function: $\Omega$- normative semantic function, or $\Omega_f$ - the fault semantic function. Instead

of introducing new components into the language, such as a *late reading dynamic state* or a *value read IDA fault* components, we instead rely on the implementation –or application– of $\Omega$ and $\Omega_f$ to be selective. We indicate in the specification which components we expect to be faulty by specifying speculative protocols or fault transitions on them. It is this observation that determines if the component is defined by $\Omega$ or $\Omega_f$

As an aid to the reader and for clarity to show the extensions from $\Omega$ to $\Omega_f$ we present $\Omega_f$ in the same structure as [Pay02]. That is, $\Omega_f$ can be seen as a collection of functions:

- $\Omega_{SM_f}$ for converting the activity state-machine into PVS,
- $\Omega_{Op_f}$ for converting RTN-SL operations and functions into PVS.

Note, $\Omega_{IDA}$ is also a function that composes $\Omega_f$, but are unchanged from $\Omega$.

#### 4.3.2.1 State-Machines, $\Omega_{SM_f}$

The function, $\Omega_{SM}$ is presented by means of a series of axiom schemata which, for each syntactic entity in the state-machine (states, transitions, etc.), specifies the exhibited dynamic behaviour. $\Omega_{SM}$ is enriched to be $\Omega_{SM_f}$ to yield an extended axiom schemata which specify the behaviour of state-machines given the definition of faults in Section 4.2.1.1.

For brevity in the axiom schema below, those existing auxiliary functions and unchanged axioms are omitted and the reader is referred to [Pay02] for those definitions. As an visual aid to the reader, we shade the new formulas and predicates we propose adding to the semantics. To make the dynamic behaviour more visible in the following presentation, the quantification over the syntactic entities of the RTN-SL specification is left implicit.

**Start-up Axiom**

It is assumed that no faults can occur before time index 0, therefore the start-up axiom is assumed to hold in $\Omega_{SM_f}$ as in $\Omega_{SM}$.

**State Exit Axioms**

**Ax 2 - Static State Exit**

$$
\begin{aligned}
&Ax\ 2\ [is_StaticState(s), TransitionsOut(s,ts) \neq 0, s_t \in Successors(s,states,ts)]\\
&\quad \forall i: Occ, t: Time \cdot \Theta(\downarrow s_t, i, t) \;\Rightarrow\\
&\qquad ((SE(s,states,ts) \wedge s \neq initial \wedge \Theta(\uparrow s_t, i, t)) \vee\\
&\qquad (SE(s,states,ts) \wedge s = initial \wedge \Theta(\uparrow s_t, i+1, t)) \vee\\
&\qquad (ME(s,states,ts) \wedge s \neq initial \wedge \exists j: Occ \cdot j \geq i \wedge \Theta(\uparrow s_t, j, t)) \vee\\
&\qquad (ME(s,states,ts) \wedge s = initial \wedge \exists j: Occ \cdot j \geq i+1 \wedge \Theta(\uparrow s_t, j, t)) \vee\\
&\qquad (\exists j: Occ \cdot \Theta(late_exit(s), j, t) \vee \Theta(early_exit(s), j, t)))
\end{aligned}
$$

We state that for each occurrence of entering a static state ($s$) we should leave it as was specified before (i.e. timely) or that a *late exit* or *early exit* fault occurs. The number of occurrences of the fault is not the same as that of entering the state, as such, faults can be observed numerous times during an interval, though we only need specify one such fault need occur.

It is worth noting the structure of Ax2 above (which is repeated throughout this presentation) that faults are seen as *alternative* behaviours in $\Omega_f$ to that that was allowed in $\Omega$. We therefore add the fault behaviours to each axiom schemata as disjunctions. This is consistent with our observation that in $\Omega$ **no faults** (or the absence thereof) is assumed, so one should have specified in $\Omega$ the explicit absence of faults.

### Ax 3 - Dynamic State Exit

The form of the existing axiom –as stated previously– is complicated by the relationship between occurrences numbers of two connected states. Furthermore, it is argued that implicit to $\Omega_{SM}$ is the absence of faults and therefore $\Omega_{SM_f}$ should explicitly state this fact to ensure the consistency of the axioms.

$Ax\ 3\ [is_dynamicState(s)]$

$$
\begin{array}{l}
\forall i : Occ, t : Time \cdot \\
\quad (\Theta(\downarrow s, i, t) \wedge \exists tr : ts \cdot tr \in Enabled(s, ts, t) \wedge \neg fault_guard?(tr.grd) \;\Rightarrow \\
\quad\quad let\ s_t = tr.tgt\ in \\
\quad\quad\quad (s = s_t \wedge \Theta(\uparrow s_t, i+1, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge SX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t \neq initial \wedge \Theta(\uparrow s_t, i, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge SX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t = initial \wedge \Theta(\uparrow s_t, i+1, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge SX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \geq i \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge SX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \geq i+1 \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge MX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \leq i \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge MX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \leq i+1 \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge MX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot \Theta(\uparrow s_t, j, t)) \vee \\
\quad\quad\quad (s \neq s_t \wedge MX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot \Theta(\uparrow s_t, j, t)) \\
\quad ) \vee \\
\quad (\exists j : Occ, tr : ts \cdot tr \in Enabled(s, ts, t) \wedge fault_guard?(tr.grd) \;\Rightarrow \\
\quad\quad\quad let\ s_t = tr.tgt\ in \\
\quad\quad\quad\quad ((tr.grd = CRASH \;\Rightarrow\; \Theta(crash_fault, j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = LATE_READ \;\Rightarrow\; \Theta(late_rd(s.in_p), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = OMIT_READ \;\Rightarrow\; \Theta(omit_rd(s.in_p), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = OMIT_WRITE \;\Rightarrow\; \Theta(omit_we(s.out_p), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = WRITE_VALUE \;\Rightarrow\; \Theta(we_value(s.out_p), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = READ_VALUE \;\Rightarrow\; \Theta(rd_value(s.in_p), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = LATE_EXIT \;\Rightarrow\; \Theta(late_exit(s), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = EARLY_EXIT \;\Rightarrow\; \Theta(early_exit(s), j, t)) \vee \\
\quad\quad\quad\quad (tr.grd = LATE_WRITE \;\Rightarrow\; \Theta(late_we(s.out_p), j, t))) \wedge \\
\quad (s \neq s_t \wedge SX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t \neq initial \wedge \Theta(\uparrow s_t, i, t)) \vee \\
\quad (s \neq s_t \wedge SX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t = initial \wedge \Theta(\uparrow s_t, i+1, t)) \vee \\
\quad (s \neq s_t \wedge SX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \geq i \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad (s \neq s_t \wedge SX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \geq i+1 \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad (s \neq s_t \wedge MX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \leq i \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad (s \neq s_t \wedge MX(s, states, ts) \wedge SE(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot j \leq i+1 \wedge \Theta(\uparrow s_t, j, t)) \vee \\
\quad (s \neq s_t \wedge MX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t \neq initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot \Theta(\uparrow s_t, j, t)) \vee \\
\quad (s \neq s_t \wedge MX(s, states, ts) \wedge ME(s_t, states, ts) \wedge s_t = initial \wedge \\
\quad\quad\quad\quad \exists j : Occ \cdot \Theta(\uparrow s_t, j, t))) \\
\quad )
\end{array}
$$

As before, the fault behaviours are added as a disjunction to the non-faulty behaviour previously defined in $\Omega$. However, this presentation is complicated with the intricacies between the source and target states of a transition ($tr$), such that the expected behaviour is determined by the type of state a transitions source and target is. However, it is obvious to see that for each fault considered, a relevant event must be observable to allow the transition (guard) to be satisfied.

**State Entry Axioms** It is perceived that faults are handled when exiting states, such that the only modification to Ax 4 & Ax 5 would be the conjunction that no faults have occurred, which is implicit to $\Omega_{SM}$.

**State Progress Axioms** $\Omega_{SM}$ states "*Dynamic states are always exited within their time bounds*" as the implicit assumption no timing faults can occur, yet given the definition of faults in Section 4.2.1.1 we propose the following modifications:

$$
\begin{array}{l}
Ax\ 6\ [is_dynamicState(s)] \\
\quad \forall i: Occ, t: Time \cdot \Theta(\uparrow s, i, t) \Rightarrow \\
\quad\quad (\exists t_1 : Time \cdot (t + s.timing.WCRT \leq t_1 \leq t + s.timing.BCET \wedge \Theta(\downarrow s, i, t_1)) \vee \\
\quad\quad\quad\quad \exists j: Occ \cdot (\Theta(crash(s), j, t_1) \vee \\
\quad\quad\quad\quad\quad\quad \Theta(late_exit(s), j, t_1) \vee \\
\quad\quad\quad\quad\quad\quad \Theta(early_exit(s), j, t_1)) \wedge \\
\quad\quad\quad\quad\quad\quad t_1 > t \\
\quad\quad )
\end{array}
$$

Ax 7 holds *per se* as faults are made events in $\Omega_{SM_f}$.

Ax 8 is modified similarly to Ax 6 by weakening the behaviour to allow for the occurrence of faults:

$$
\begin{array}{l}
Ax\ 8\ [is_staticState(s), is_TimeBounds(tr.label), tr.label.x = (l, u)] \\
\quad \forall i: Occ, t: Time \cdot \Theta(\uparrow s, i, t) \Rightarrow \\
\quad\quad (\exists t_1 : Time \cdot (t + l \leq t_1 \leq t + u \wedge \Theta(\downarrow s, i, t_1)) \vee \\
\quad\quad\quad\quad \exists j: Occ \cdot (\Theta(early_exit(s), j, t_1) \vee \\
\quad\quad\quad\quad\quad\quad \Theta(late_exit(s), j, t_1)) \wedge \\
\quad\quad\quad\quad\quad\quad t_1 > t \\
\quad\quad )
\end{array}
$$

The modification to Ax 6 & Ax 8 simply allow the possibility a fault has occurred, rather than specify the behaviour exhibited. The fault behaviours themselves are defined in other axioms. These modifications are required to ensure the PVS model is consistent.

**State Stability Axioms** Given the approach that faults are characterised on exit from states, then the stability axioms (namely Ax 9 & Ax 10) are largely unaffected in $\Omega_{SM_f}$, other than to weaken the assertion to include the possibility of faults occurring.

$Ax$ 9 [$is_DynamicState(s)$]

$$\forall i : Occ, t : Time \cdot \Theta(\downarrow s, i, t) \Rightarrow$$
$$(\exists t_1 : Time \cdot (t\text{-}s.timing.WCRT \leq t_1 \leq t\text{-}s.timing.BCET \wedge \Theta(\uparrow s, i, t_1)) \vee$$
$$\exists j : Occ \cdot (\Theta(early_exit(s), j, t_1) \vee$$
$$\Theta(late_exit(s), j, t_1)) \wedge$$
$$t_1 > t$$
$$)$$

$Ax$ 10 [$is_StaticState(s), \#Successor(s, states, ts) \neq 0, is_TimeBounds(tr.label), tr.label.x = (l, u)$]

$$\forall i : Occ, t : Time \cdot \Theta(\downarrow s, i, t) \Rightarrow$$
$$(\exists t_1 : Time \cdot (t\text{-}u \leq t_1 \leq t\text{-}l \wedge \Theta(\uparrow s, i, t_1)) \vee$$
$$\exists j : Occ \cdot (\Theta(early_exit(s), j, t_1) \vee$$
$$\Theta(late_exit(s), j, t_1)) \wedge$$
$$t_1 > t$$
$$)$$

Ax 11 & Ax 12 as defined in $\Omega_{SM}$.

#### 4.3.2.2 Operations, $\Omega_{Op_f}$

Ax 13 asserts the post condition of an operation must hold for the input and output parameter read and written respectively.

$Ax$ 13 [$is_DynamicState(s)$]

$$\forall i : Occ, t_1, t_2 : Time \cdot \Theta(\uparrow s, i, t_1) \wedge \Theta(\downarrow s, i, t_2) \Rightarrow$$
$$(post_s(p_in(t_1), \overleftarrow{v}(t_1), v(t_2), p_out(t_2)) \wedge \Omega_{Frame}) \wedge$$
$$\nexists j : Occ \cdot \Theta(late_rd(s.in_p), j, t_1) \wedge$$
$$\Theta(rd_value(s.in_p), j, t_1) \wedge$$
$$\Theta(late_we(s.out_p), j, t_2) \wedge$$
$$\Theta(we_value(s.out_p), j, t_2)$$

# Chapter 5

# Structural Operational Semantics

## Contents


**5.1 Basics** . . . . . . . . . . 98

5.1.1 Labelled Transition Systems . . . . . . . . . . 98

5.1.2 Plotkin-style Transition Rules . . . . . . . . . . 100

**5.2 An Operational Model for RTNs** . . . . . . . . . . 100

5.2.1 RTN-SL SOS Rules . . . . . . . . . . 104

**5.3 An Operational Conservative Extension** . . . . . . . . . . 114

5.3.1 Definitions . . . . . . . . . . 114

5.3.2 Faults are a conservative extension . . . . . . . . . . 116

5.3.3 Animating SOS rules with LETOS . . . . . . . . . . 117


We set up a formal framework to describe transition system specifications in the style of Plotkin known as *Structural Operational Semantics* (SOS) [Plo81]. The framework is used to present a conservativity format in operational semantics, which states sufficient criteria to ensure that the extension of a transition system specification with new transition rules does not affect the semantics of the original terms.

Before presenting the Plotkin-style rules for RTNs, we first introduce some syntactic sugar which is later required to show the SOS rules for faults are a conservative extension of the non-fault transition system. In Section 5.2 we present the SOS rules which define the semantics of activities in an RTN and the faults feasible. Section 5.3 first introduces the theory of a conservative extension and states that our fault semantics are such an extension. Section C.5 discusses the issue of concurrency and non-determinism in transition systems. This relates to expressing the "meta-level" specification over the SOS rules, which specifies how each rule *fires* in parallel with other rules and which rule should fire next. We conclude with an investigation into implementing the SOS rules and discuss the implications with regards non-determinism.

## 5.1 Basics

### 5.1.1 Labelled Transition Systems

What follows is a description of the model of *labelled transition systems* (LTS) as described in [AFV99], which are used in turn to express the operational semantics of RTNs. We later establish a notion of a "meta-level" which is a specification of the LTS with regards our RTN semantic rules.

A LTS consists of binary relations between states, labelled with an action as a predicate between its states. Intuitively, $s \xrightarrow{a} s'$ expresses that state $s$ can transition to state $s'$ through the execution of action $a$.

**Definition 5.1 (Labelled Transition System)** *A labelled transition system (LTS) is a triple* $(Proc, Act, \{\xrightarrow{a} \mid a \in Act\})$*, where:*

- *Proc is a set of states, ranged over by* $s, s'$*;*
- *Act is a set of actions, ranged over by* $a, b$*;*
- $\xrightarrow{a} \subseteq Proc \times Proc$ *for every* $a \in Act$*. As usual, we use the more suggestive notation* $s \xrightarrow{a} s'$ *in lieu of* $(s, s') \in \xrightarrow{a}$*, and write* $s \overset{a}{\not\rightarrow}$ *if* $s \xrightarrow{a} s'$ *for no state* $s'$*;*

Binary relations $s \xrightarrow{a} s'$ in a LTS are called *transitions.*

**Definition 5.2 (Trace Semantics)** *Given a LTS* $(Proc, Act, \{\xrightarrow{a} \mid a \in Act\})$*, a sequence*

$$\varsigma = a_1, \ldots, a_n \in Act^*$$

*for* $n \in \mathbb{N}$ *is said to be a trace of state* $s_0$ *if there exist states* $s_1, \ldots, s_n$ *such that* $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ *(abbr. by* $s_0 \xrightarrow{\varsigma} s_n$*).*

We later embrace Definition 5.2 in our choice of trace model for RTNs, which essentially is an ordered sequence of Event sets.

#### 5.1.1.1 Term Algebras

This section reviews some basic notions of term algebras that will be needed later in the chapter.

**Definition 5.3 (Signature)** *A signature* $\Sigma$ *is a set of function symbols together with an arity mapping that assigns a natural number* $ar(f)$ *to each function symbol* $f$*. A function symbol of arity zero is called a constant, while function symbols of arity one and two are called unary and binary, respectively.*

**Definition 5.4 (Term)** *The set* $\mathbb{T}$ *of (open) terms over a signature* $\Sigma$ *denoted* $\mathbb{T}(\Sigma)$*, ranged over by* $t$ *is the least set such that:*

- *each* $x \in Var$ *is a term;*
- $f(t_1, \ldots, t_{ar(f)})$ *is a term, iff* $f$ *is a function symbol in* $\Sigma$ *and* $t_1, \ldots, t_{ar(f)}$ *are terms.*

$\mathrm{T}(\Sigma)$ denotes the set of closed terms over $\Sigma$, i.e., terms that do not contain variables.

#### 5.1.1.2 Transition System Specifications

**Definition 5.5 (Transition Rule)** *Let $\Sigma$ be a signature, and let $t$ and $t'$ range over $\mathbb{T}(\Sigma)$. A transition rule $\rho$ is of the form $\frac{H}{\alpha}$, with $H$ a set of premises of the form $t \xrightarrow{a} t'$ or $tP$ and a conclusion $\alpha$ of the form $t \xrightarrow{a} t'$ or $tP$. The left-hand side of the conclusion is the source of $\rho$, and if the conclusion is of the form $t \xrightarrow{a} t'$, then is its right-hand side is the target of $\rho$. A transition rule is closed if it does not contain any variables.*

**Definition 5.6 (Transition System Specification)** *A transition system specification (TSS) is a set of transition rules.*

**Definition 5.7 (Proof)** *Let $T_a$ be a TSS. A proof of a closed transition rule $\frac{H}{\alpha}$ from $T_a$ is a well-formed, upwardly branching derivation tree whose nodes are labelled by transitions, where the root is labelled by $\alpha$, and if $K$ is the set of labels of the nodes directly above a node with label $\beta$, then*

1. *either $K = \{\}$ and $\beta \in H$,*
2. *or $\frac{K}{\beta}$ is a closed substitution of a transition rule in $T_a$.*

*If a proof of $\frac{H}{\alpha}$ from $T_a$ exists, then $\frac{H}{\alpha}$ is provable from $T_a$, notation $T_a \vdash \frac{H}{\alpha}$.*

### 5.1.2 Plotkin-style Transition Rules

Structural operational semantics (SOS) [Plo81, Plo03] provides a framework for giving an operational semantics of programming and specification languages. In particular, because of its intuitive appeal and flexibility, SOS has found considerable application in the study of semantics of concurrent processes [Mil80, ABV94]. The SOS rules of a language generates a labelled transition system, whose states are the closed terms over an algebraic signature, and whose transitions are supplied with labels. The transitions between states are obtained inductively from a transition system specification (TSS), which consists of transition rules of the form $\frac{premises}{conclusions}$. A typical example of a transition rule is

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y}$$

stipulating that if $x \xrightarrow{a} x'$ holds for closed terms $x$ and $x'$, then so does $x \| y \xrightarrow{a} x' \| y$ for any closed term $y$. Each rule is then a label on transitions in the LTS theoretical framework. An important point about SOS is that it is structural, not, structured. The idea is that, in denotational semantics one follows an idea of compositionality, where the meaning of a compound phrase is given as a function of the meaning of its parts. In the case of operational semantics the behaviour of the program is, roughly speaking, the collection of transitions it can make [Plo03].

## 5.2 An Operational Model for RTNs

We wish to give a structural operational semantics for RTN-SL, which consists of the possible transitions, or actions, a RTN can make. A SOS framework is one which plots out each transition in a RTN semantic

model. The values between these transitions are referred to as *configurations* [Win93], to differentiate them from the dynamic state of an RTN, which is formed from values of variables within an RTN –such as local variables in activities and internal variables in IDAs– and the (semantic) model states. Given a (static) RTN specification, then considering its dynamic behaviour is to consider the transitions from configuration to configuration. A configuration is a triple $(RTN : RTN\text{-}Types, \sigma_1 : \Sigma, \pi_1 : \Pi)$, where

$$RTN\text{-}Types = Id \xrightarrow{m} (Activity \mid IDA \mid Connection)$$
$$\Sigma = Id \xrightarrow{m} dyTypes$$
$$\Pi = Event\text{-set}^*$$

The *RTN-Types* element represents the abstract syntax of a static RTN-SL specification, $\Sigma$ the dynamic values of those components in an RTN which have *dynamic values* (for example local variables and current state registers) and $\Pi$ is the history -or trace- of an RTN to date. The trace, $\pi_1$ is a sequenced set of (observable external) events, indexed by time. The abstract syntax and static semantics for the rules that follow, may be found in Appendix C.1. The VDM-SL model used to facilitate syntax and type checking the SOS rules and provide a degree of animation is discussed later in Section C.5.1.

The operational semantics of RTNs begins with the idea that the dynamic behaviour of an RTN is the sequence of 'RTN Actions' and the semantics thereof the actions. Several *actions* are permissible in an RTN. These are:

- read and write actions of IDAs and the subsequent actions withing activities;
- state-machine transitions from source states to targets.

Each of these actions are expressed atomically by disjoint SOS rules. An action will generate an event such as $\Uparrow s$ for entering a state $s$, or $rp_p1$ for reading a port $p1$. An action may also modify the dynamic values within an activity or IDA, e.g. by reading from or writing to an IDA. Considering the idea that RTN actions will transform the state, we consider the function:

$$\xrightarrow{s} : Action \times \Sigma \rightarrow \Sigma$$

However, this recursive function style approach –although intuitive– does not handle non-determinism and concurrency. The evaluation relation

$$(rtn, \sigma, \pi) \rightarrow \pi'$$

specifies the evaluation of an RTN in rather large steps; given the static specification and initial values then yield a final trace directly. This choice of *large* step semantics makes the issue of parallelism and non-determinism opaque and hides the non-deterministic choices deep within the rules. Rather, a *small* step semantic is preferred which makes it possible to give rules for evaluation which capture the single steps. We therefore (formally) define an evaluation relation between *pairs of configurations*. We therefore could define a relation as:

$$\xrightarrow{s} : ((RTN\text{-}Types \times \Sigma \times \Pi) \times (\Sigma \times \Pi))$$

Although this $\xrightarrow{s}$ relation captures the semantics of the set of actions occurring, it does not provide for the ramification of this powerset. From a configuration, many concurrent actions can, and should, fire - yet the *set of* states should be ramified to yield the next configuration.

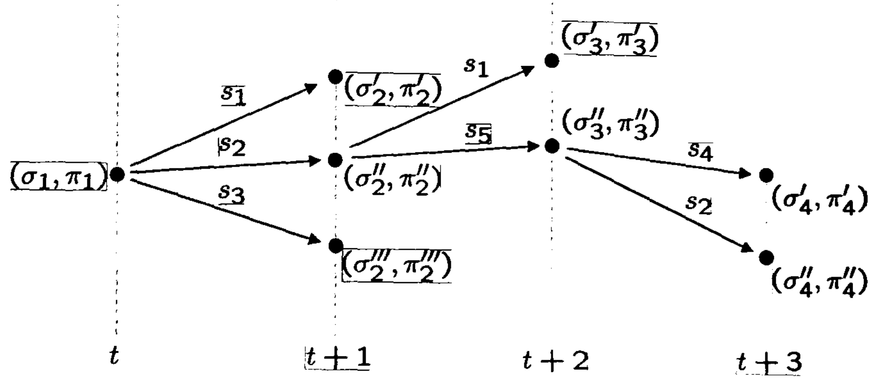



Figure 5.1: Possible derivation tree from some starting configuration, $(\sigma_1, \pi_1)$

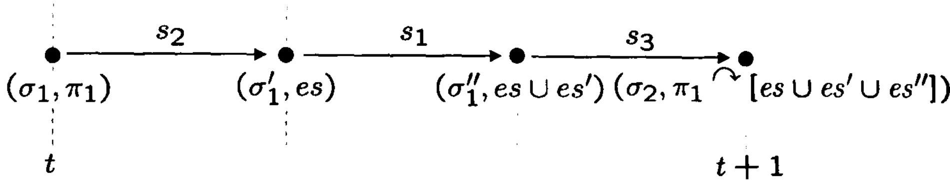



Figure 5.2: Possible sequencing of actions

Figure 5.2 illustrates the divergence of configurations from a starting configuration when two or more actions can fire. This possible derivation tree shows the generation of new configurations having executed each action, whereas we require to ramify these resulting values for each step. It can be see from Figure 5.1 that a transition from a configuration to another is autonomous to a step in time. Therefore, if -say- action $s_1$ is evaluated from a starting configuration $(\sigma_1, \pi_1)$, then actions $s_2$ and $s_3$ may not still be valid in $(\sigma'_2, \pi'_2)$, if only that time was advanced! Considering the implications on ramifying $\sigma'_2$, $\sigma''_2$ and $\sigma'''_2$ is technically challenging as it is unknown which variables of $\sigma^x_2$ have been modified. Ramifying $\pi^x_2$ is straightforward, it is simply the union of the events at each time instance. The observation that ramifying each resulting $\pi^x_2$ is straightforward suggests the event-set generated by each action are disjoint, which is as expected given actions are non-interfering and atomic. It is therefore feasible for each action to *fire* from the resulting state $(\sigma^x_2)$ of another action **within** the same time instance, as illustrated in Figure 5.2. We then take the union of each event set and record within the trace at $t+1$.

The transition semantics are therefore expressed in a two-tier approach. The first is dedicated to expressing the non-deterministic, concurrent semantics of a single transition, which equates to one time unit (in our discrete time model). The second tier expresses the small-step semantics of an RTN action. Given RTNs are truly concurrent, then so too are its actions. An action should only *fire* in a given configuration if the premise(s) of its associated SOS rule hold true. Given that actions can fire concurrently, the events generated in one action **cannot** interfere with another firing from the same configuration.

Conclusively, we propose to specify the operational semantics for RTN-SL by two relations:

$$\xrightarrow{rtn} :((\textit{RTN-Types} \times \Sigma \times \Pi) \times (\Sigma \times \Pi))$$
$$\xrightarrow{s} :((\textit{RTN-Types} \times \Sigma \times \Pi) \times (\Sigma \times \textit{Event}\text{-set}))$$

The evaluation relation $\xrightarrow{s}$ of a *small* (atomic) step is one which generates a new state ($\Sigma$) and an event set (*Event*-set) - the observable events generated in the transition. The first tier is expressed by two SOS rules that specify the $\xrightarrow{rtn}$ relation. The two rules specify a single step of an RTN, such that each concurrent component in an RTN take a small step from a configuration. Once no further rules in $\xrightarrow{s}$ can fire from a configuration, the time base -the indices of $\pi_1$ - is advanced. Consecutive 'tick' events are permitted, which would mean the first rule is evaluated in successive configurations.

The second $\xrightarrow{rtn}$ rule specifies that each rule in $\xrightarrow{s}$ that can fire does so, such that the resulting state ($\sigma_1'$) is carried forward for the recursive premise. The event set ($es'$) produced from the $\xrightarrow{s}$ action are *collected*. Note, events occurring at time, $t$ cannot affect the semantics of other actions firing at $t$. Until there are no further small step actions that can fire from an intermediate configuration, we collect each event set ($es$) generated in a *handbag* which is committed to the trace ($\pi_1 \frown [es, \{\}]$) in the *Tick* $\xrightarrow{rtn}$ rule which records these events in our trace and increments the time base by one, as was illustrated in Figure 5.2.

$$\boxed{\text{tick}}\ \frac{(rtn, \sigma_1, \pi_1) \not\xrightarrow{s}}{(rtn, \sigma_1, \pi_1 \frown [es]) \xrightarrow{rtn} (\sigma_1, \pi_1 \frown [es, \{\}])}$$

$$\boxed{\text{step}}\ \frac{(rtn, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1', es') \quad (rtn, \sigma_1, \pi_1 \frown [es \cup es']) \xrightarrow{rtn} (\sigma_2, \pi_2)}{(rtn, \sigma_1, \pi_1 \frown [es]) \xrightarrow{rtn} (\sigma_2, \pi_2)}$$

Described in these rules are three key points:

1. "$(rtn, \sigma_1, \pi_1 \frown [es]) \xrightarrow{rtn} \ldots$" :: The form of the *source* of both rules explicitly separates the trace -or history- of an RTN into the events upto $t$-1 and those events occurring at $t$. This is to enforce atomicity and model currency such that $A \parallel B \equiv A;B$ Or $B;A$

2. "$\ldots \xrightarrow{rtn} (\sigma_1, \pi_1 \frown [es, \{\}])$" :: The form of the target of the first rule is such that those events occurring at $t$ are *committed* to the trace of an RTN and time is advanced

3. The observation that these rules describe a *greedy* semantics. Each rule that can fire in a configuration must do so before time is advanced. The benefit of this is that *maximal* progress is made in each time-step. This design decision is consistent with the STATECHART semantics [HN96] which specify a maximal subset of non-conflicting transitions are always executed which is termed the "greediness property" of the semantics.

### 5.2.1 RTN-SL SOS Rules

We first give a transition system specification ($T_0$), for the $\xrightarrow{s}$ relation, for the semantics of RTN-SL. Later we give the TSS for the extended RTN-SL language which supports the specification of faults.

#### 5.2.1.1 A TSS Specification for RTNs, $T_0$

The set of rules that follow form the transition system specification for $T_0$. These rules are grouped (indicated by number suffixes) when a rule describes a similar dynamic behaviour to another rule. In such cases a generic description is given for the group with individual comment to highlight the distinction.

We present the RTN-SL abstract syntax on which the SOS rules are written, to aid the reader. This approach is not dissimilar to other presentation of language semantics [Jon03, Jon04]. The abstract syntax is given in the VDM-SL notation [LHP+96, FL98], and each SOS rule written as a VDM-SL function in Appendix C to syntax and type check the rules. Additionally, the implementation issues of the operational semantics with regards concurrency and non-determinism is investigated and reported with regards animation in Appendix point 3 above.

types

1.0 $Id$ = token;

2.0 $Var$ = token;

3.0 $Expr$ = token;

4.0 $Event$ = token;

5.0 $Fault$ = LATE_EXIT;

6.0 $Bounds :: bcet : \mathbb{N}$
.1 $bcrt : \mathbb{N}$
.2 $wcet : \mathbb{N}$;

7.0 $Port :: id : Id$
.1 $dir$ : IN | OUT;

8.0 $TimeBound :: lower : \mathbb{N}$
.1 $upper : \mathbb{N}$;

9.0 $Label = TimeBound \mid Event \mid Expr \mid Fault$;

10.0 $Static\text{-}State :: id : Id$;

The first group of rules describes the reactive behaviour of dynamic states - should a state read, write, both or neither? The necessary abstract syntax is shown below:

11.0 *Operation* :: *id* : *Id*
.1 *read-vars* : *Var*-set
.2 *written-vars* : *Var*-set
.3 *input-parameter* : [*Id*]
.4 *output-result* : [*Id*]
.5 *preC* : *Expr*
.6 *postC* : *Expr*;

12.0 *Dynamic-State* :: *id* : *Id*
.1 *ops* : *Operation*-set
.2 *bounds* : *Bounds*;

13.0 *State* = *Static-State* | *Dynamic-State*;

14.0 *Transition* :: *src* : *Id*
.1 *trg* : *Id*
.2 *l* : *Label*;

The premises of Rule 1.1 identify that there exists an activity ($\sigma_1(a) = mk_act(cs, ls, \text{INDS})$) in the RTN dynamic state ($a \in \text{dom}\,\sigma_1$) which satisfies three conditions: i) the current state, $cs$ has operations remaining (line 3), ii) that the operation (and therefore state) reads from a port but does not write a result (line 6), and iii) lines 6-10 state that the *pre*-condition holds for the read value, and the *post*-condition is **true** in the new local state, $ls'$. Given these premises hold true in a configuration $(RTN, \sigma_1, \pi_1)$, then the conclusion is that the resulting state of this transition has the updated dynamic state (removing the evaluated operation $\sigma_1 \dagger \{cs \rightarrow \text{tl}\,\sigma_1(cs).ops\}$) and the event $(rd(in_p))$ recorded.

Rule 1.1: A dynamic state that only reads:

$$\frac{\begin{array}{l} a \in \text{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ \sigma_1(cs).ops \neq [\,] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \text{elems}\, s.ops \wedge \text{hd}\,\sigma_1(cs).ops = id \\ in_p \neq \textbf{nil}, out_p = \textbf{nil} \\ (in_p, RTN, \sigma_1) \xrightarrow{rd} v \\ ls' = ls \dagger \{in_p \mapsto v\} \\ (pre, ls) \xrightarrow{e} \textbf{true} \\ (post, ls') \xrightarrow{e} \textbf{true} \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \text{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls', \text{INDS})\}, \{rd(in_p)\})}\ \text{[dySt-op-rd]}$$

Similarly with Rules 1.2, 1.3 and 1.4, the premises are that the written values preserves the *post*-condition in Rule 1.2 and Rule 1.3 and that Rule 1.4 neither reads nor writes but the *post*-condition holds on the computation of the local state, $ls$.

Rule 1.2: A dynamic state that reads and writes:

$$\boxed{\text{dySt-op-rdwe}}\ \frac{\begin{array}{l}a \in \mathsf{dom}\,\sigma_1\\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS})\\ \sigma_1(cs).ops \neq [\,]\\ s \in RTN(a).sm.ss\\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathsf{elems}\, s.ops \wedge \mathsf{hd}\,\sigma_1(cs).ops = id\\ in_p \neq \mathbf{nil}, out_p \neq \mathbf{nil}\\ (in_p, RTN, \sigma_1) \xrightarrow{rd} v\\ ls' = ls \dagger \{in_p \mapsto v\}\\ (pre, ls) \xrightarrow{e} \mathbf{true}\\ (out_p, v', ls') \xrightarrow{we} ls''\\ (post, ls'') \xrightarrow{e} \mathbf{true}\end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathsf{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls'', \mathrm{INDS})\}, \{rd(in_p), we(out_p)\})}$$

Rule 1.3: A dynamic state that only writes:

$$\boxed{\text{dySt-op-we}}\ \frac{\begin{array}{l}a \in \mathsf{dom}\,\sigma_1\\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS})\\ \sigma_1(cs).ops \neq [\,]\\ s \in RTN(a).sm.ss\\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathsf{elems}\, s.ops \wedge \mathsf{hd}\,\sigma_1(cs).ops = id\\ in_p = \mathbf{nil}, out_p \neq \mathbf{nil}\\ (out_p, v', ls) \xrightarrow{we} ls'\\ (post, ls') \xrightarrow{e} \mathbf{true}\end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathsf{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls', \mathrm{INDS})\}, \{we(out_p)\})}$$

Given the SOS rule that an operation specified upon a dynamic state neither reads nor writes, we note the local state (*ls*) is unchanged by the transition. Since post conditions are expressions not statements, they can have no side effect on the local state and can only specify an expression on the state.

Rule 1.4: A dynamic state that neither reads nor writes:

$$\boxed{\text{dySt-op}}\ \frac{\begin{array}{l}a \in \mathsf{dom}\,\sigma_1\\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS})\\ \sigma_1(cs).ops \neq [\,]\\ s \in RTN(a).sm.ss\\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathsf{elems}\, s.ops \wedge \mathsf{hd}\,\sigma_1(cs).ops = id\\ in_p = \mathbf{nil}, out_p = \mathbf{nil}\\ (pre, ls) \xrightarrow{e} \mathbf{true}\\ (post, ls) \xrightarrow{e} \mathbf{true}\end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathsf{tl}\,\sigma_1(cs).ops\}, \{\,\})}$$

The second group of rules define the behaviour of activities should the initial state be static or dynamic. For a static state, the activity's status is simply set to the *wait transition* (WAIT_ TR) status which stipulates the activity is waiting for a valid exit transition, for example, a read event to occur at a blocking (input) port which is the guard of an exit transition. Alternatively -for a dynamic state- the activity is set to the *in-dynamic-state* (INDS) status and the operations specified upon it in the static specification (*RTN*) are

added to the dynamic environment (Line 5) for evaluation. For both cases, the state entry event is recorded ($\Uparrow$*initial*).

15.0 $SM :: ss : State\text{-set}$
.1 $ts : Transition\text{-set}$
.2 $initial : Id;$

16.0 $Activity :: input\text{-}ports : Port\text{-set}$
.1 $output\text{-}ports : Port\text{-set}$
.2 $local\text{-}state : Id \xrightarrow{m} Var$
.3 $ops : Operation\text{-set}$
.4 $sm : SM;$

Rule 2.1: Initialise an activity into a static state:

$$\text{[stSt-init]}\ \frac{\begin{array}{l} a \in \text{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(_,_,\text{INIT}) \\ uact = mk_act(RTN(a).sm.initial, RTN(a).local\text{-}state, \text{WAIT_TR}) \end{array}}{(RTN,\sigma_1,\pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\Uparrow RTN(a).sm.initial\})}$$

Rule 2.2: Initialise an activity into a dynamic state:

$$\text{[dySt-init]}\ \frac{\begin{array}{l} a \in \text{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INIT}) \\ uact = mk_act(RTN(a).sm.initial, RTN(a).local\text{-}state, \text{INDS}) \\ mk_Dynamic_State(id, ops, _) \in RTN(a).sm.ss \wedge id = cs \\ trgs = mk_dySt([op \mid op \in ops]) \end{array}}{\begin{array}{l}(RTN,\sigma_1,\pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, RTN(a).sm.initial \mapsto trgs\}, \\ \qquad \{\Uparrow RTN(a).sm.initial\})\end{array}}$$

Rule 3.1 specifically defines that a dynamic state (Line 2) with no remaining operations left to evaluate (Line 3) should set the activity status to WAIT_ TR (Line 4).

Rule 3.1: All operations on a dynamic state evaluated:

$$\text{[dySt]}\ \frac{\begin{array}{l} a \in \text{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ \sigma_1(cs).ops = \{\} \\ uact = mk_act(cs, ls, \text{WAIT_TR}) \end{array}}{(RTN,\sigma_1,\pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\})}$$

Rule 4.1 defines when an activity is terminated which detects the end of a network's computation (all activities are TERM).

Rule 4.1: Terminate an activity:

$$\boxed{\text{term}}\ \frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 \\ is_(\sigma_1(a), act) \\ \nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(\sigma_1(a).cs, \sigma_1(a).ls, \text{TERM})\}, \{\})}$$

Rules 5.1 & 5.2 both depend on whether the target state of a transition is a dynamic or static state. Should the target state be dynamic, then the operations specified on the dynamic state should be added to the dynamic environment for evaluation, as was the case in Rule 2.2.

Rule 5.1: Transition to dynamic state:

$$\boxed{\text{tr-dySt}}\ \frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{WAIT_TR}) \\ tr \in RTN(a).sm.ts \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ mk_Dynamic_State(id,, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow cs \in_i \pi_1(t_1) \wedge t_1 + bcet \leq \mathsf{len}\,\pi_1 < t_1 + wcet \\ \qquad\qquad \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot t_2 \geq t_1 \wedge \uparrow cs \in_{i+1} \pi_1(t_2) \\ ins(\sigma_1(tr.trg)) = \text{INDS} \\ uact = mk_act(tr.trg, ls, \text{INDS}) \\ trgs = mk_dySt([op \mid op \in s.ops \cdot s \in RTN(a).sm.ss \wedge s.id = tr.trg]) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, tr.trg \mapsto trgs\}, \{\Downarrow tr.src, \Uparrow tr.trg\})}$$

Similarly, should the target state be static, the activity should be set to wait for a valid transition from this target state.

Rule 5.2: Transition to static state:

$$\boxed{\text{tr-stSt}}\ \frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{WAIT_TR}) \\ tr \in RTN(a).sm.ts \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ mk_Dynamic_State(id,, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow cs \in_i \pi_1(t_1) \wedge \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot t_2 \geq t_1 \wedge \uparrow cs \in_{i+1} \pi_1(t_2) \wedge \\ \qquad\qquad t_1 + bcet \leq \mathsf{len}\,\pi_1 < t_1 + wcet \\ ins(\sigma_1(tr.trg)) = \text{INSS} \\ uact = mk_act(tr.trg, ls, \text{WAIT_TR}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\Downarrow tr.src, \Uparrow tr.trg\})}$$

Rules 6.1, 6.2 & 6.3 define a new relation $(tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbb{B}$ which evaluates whether or not a transition $(tr)$ is valid from a configuration $(tr, \sigma_1, \pi_1)$. This relation has a similar role to evaluating an arithmetic operation in a programming language semantics.

Rule 6.1 evaluates a transition which is bound by lower & upper time bounds. Line 2 stipulates that from entering the state at $t_1$ then it is still possible to leave with the bounds, whereas Line 3 stipulates that the state exit event has not occurred before the lower bound nor later than the upper bound.

Rule 6.1: A time-triggered transition:

$$\boxed{\text{tt-tr}}\ \frac{\begin{array}{l} is_(tr.l, TimeBound) \\ \exists t_1 \in \text{inds}\,\pi_1 \cdot \uparrow src \in_i \pi_1(t_1) \wedge l.lower \leq \text{len}\,\pi_1\text{-}t_1 \leq l.upper \\ \nexists t_2 \in \text{inds}\,\pi_1 \cdot \downarrow trg \in_i \pi_1(t_2) \wedge l.upper \leq \text{len}\,\pi_1\text{-}t_2 \leq l.lower \end{array}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true}}$$

Rule 6.2 evaluates that a transition labelled with an event is valid in a given $(tr, \sigma_1, \pi_1)$. That is, that the labelled event has occurred since entering the state.

Rule 6.2: An event-triggered transition:

$$\boxed{\text{ev-tr}}\ \frac{\begin{array}{l} is_(tr.l, Event) \\ \exists t_1, t_2 \in \text{inds}\,\pi_1 \cdot \uparrow src \in_i \pi_1(t_1) \wedge tr.l \in \pi_1(t_2) \wedge t_2 \geq t_1 \end{array}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true}}$$

Rule 6.3 evaluates an expression, which labels a transition in $\sigma_1$.

Rule 6.3: An expression-triggered transition:

$$\boxed{\text{expr-tr}}\ \frac{\begin{array}{l} is_(l, Expression) \\ (l, \sigma_1) \xrightarrow{e} \mathbf{true} \end{array}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true}}$$

Additional SOS rule for the operational semantics for IDA components are given in Appendix C, these rules also define $T_0$ but are not presented here to aid presentation.

#### 5.2.1.2 Fault Transition System Specification for RTNs, $T_1$

We have a transition system specification, $T_0$ which defines the operational semantics for non-faulty RTNs. We now present the operational semantics for the fault actions we consider for RTNs, which form the transition system specification, $T_1$. To preserve the existing semantics we wish to add the operational semantics of faults in a conservative approach, such as to define the new RTN actions as additional behaviours. We define a new fault relation as:

$$\xrightarrow{f} : ((RTN\text{-}Types \times \Sigma \times \Pi) \times (\Sigma \times Event\text{-set}))$$

This relation defines the fault actions of an RTN, which are for example taking a *late read transition*[1] upon a late read fault, or taking a *late exit transition* when observing a late exit fault. However, given the trace semantics for RTNs are specified by the $\xrightarrow{rtn}$ relation (which in turn is specified as the small-steps under the $\xrightarrow{s}$ relation) we must define new $\xrightarrow{s}$ SOS rules to add the fault action to the operational semantics. This is achieved by specifying, as premises, the hypothesis a fault action occurs from some faulty configuration. A typical rule structure would be:

$$\boxed{\text{fault-rule}}\ \frac{\begin{array}{l} \text{fault configuration hypotheses} \\ \text{fault action: } (RTN, \sigma_1, \pi_1) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}$$

[1]Remember state-machine transitions are the primary specification mechanism for defining faults and fault action behaviours

The *fault configuration hypotheses* will *observe* a fault configuration such that some component is in an (anticipated) erroneous state. Given this observation, the *fault action hypothesis* specifies some fault action occurs which generates a new state ($\sigma_2$) and an event set ($es$). The proviso that each new SOS rule under the $\xrightarrow{s}$ relation has a premise of the new *-fresh-* relation $\xrightarrow{f}$ ensures our conservative extension result (c.f. Section 5.3). Further, is is not possible to observe a fault unless a fault action exists to deal with the fault as no rule in $\xrightarrow{s}$ will be fireable. Therefore the set of $\xrightarrow{f}$ rules define the fault behaviours that exist in the operational model.

Rule 7.1 describes the *late exit* behaviour from a dynamic state. This rule has two noticeable characteristics: Lines 1-4 *observe* the fault occurrence, whilst Line 6 specifies what fault action to take. This rule is not of a fault tolerant nature, but specifies the action desired when a fault is observed - otherwise our fault semantics would be totally non-deterministic!

Rule 7.1: Dynamic State, **late exit fault**:

$$
\text{dy-late-exit-fault}\quad
\frac{\begin{array}{l}
a \in \mathsf{dom}\,\sigma_1 \\
\sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\
mk_Dynamic_State(id, _, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\
\exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow cs \in \pi(t_1) \wedge t_1 + bcet \leq t_1 + wcet < \mathsf{len}\,\pi_1 \wedge \\
\qquad\qquad \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot \downarrow cs \in \pi(t_2) \wedge t_2 > t_1 \\
uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_EXIT})) \\
(RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_exit_fault\}]) \xrightarrow{f} (\sigma_2, es)
\end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}
$$

Rule 7.2: Static State, **late time exit fault**:

$$
\text{st-late-time-exit-fault}\quad
\frac{\begin{array}{l}
a \in \mathsf{dom}\,\sigma_1 \\
\sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\
\exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, mk_TimeBound(l, u)) \wedge src = cs \\
\exists t_1 \in \mathsf{inds}\,\pi \cdot \uparrow cs \in_i \pi(t_1) \wedge t_1 + l \leq t_1 + u < \mathsf{len}\,\pi \wedge \\
\qquad\qquad \nexists t_2 \in \mathsf{inds}\,\pi \cdot \downarrow cs \in_i \pi(t_2) \wedge t_2 > t_1 \\
uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_EXIT})) \\
(RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_exit_fault\}]) \xrightarrow{f} (\sigma_2, es)
\end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}
$$

Rule 7.1 *fires* on the occurrence of a *late exit* fault and specifies that there should exist a fault-labelled transition in the static specification, *RTN* for which that transition is made. Rules 7.1 and ?? combine to form the semantics of a *late exit* fault of a state-machine state.

Rule 8.1 describes the *late read* behaviour for a dynamic state. Lines 1-5 specify a late read fault has occurred if the read event has not occurred before the best case execution time (BCET) time bound. Line 6 hypotheses that a fault action will be taken to generate a new state ($\sigma_2$) and an event set ($es$). This intermediate configuration is also the result of the rule.

Rule 8.1: Dynamic State, **late read fault**:

$$\text{dy-late-read-fault}\quad \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge in_p \neq \mathbf{nil} \\ \exists t_1 \in \mathrm{inds}\,\pi_1 \cdot \upharpoonleft cs \in \pi(t_1) \wedge t_1 + bcet < \mathrm{len}\,\pi_1 \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi_1 \cdot rds(in_p) \in \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\text{LATE_READ})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_read_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}$$

Rule 9.1 states that there should exist a transition, labelled for a late read, which is taken given a late read fault is observed.

Rule 9.1: Fault Transition:

$$\text{fault-transition}\quad \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, mk_fault(f)) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, l) \wedge src = cs \wedge l = f \end{array}}{(RTN, \sigma_1, \pi_1 \curvearrowright [es]) \xrightarrow{f} (\sigma_1 \dagger \{a \mapsto mk_act(trg, ls, insf(trg))\}, es \cup \{f, \downharpoonleft cs, \upharpoonleft trg\})}$$

The remaining rules that form $T_1$ are presented in Appendix C, as the differences in each rule follow closely to the fault definitions given in Chapter 4. The important observation with respect $T_1$ is that the new rules under the $\xrightarrow{s}$ relation are specified with a *fresh* relation which guarantee the conservativity result we discuss in the next section.

## 5.3 An Operational Conservative Extension

Given the two transitional system specifications, $T_0$ and $T_1$ we consider their combination. The rules of TSS $T_1$ are an extension to the RTN-SL operational semantics so we consider the extended TSS, denoted $T_0 \oplus T_1$, as the operation extension of TSS $T_0$. A property we require to be true, is that the TSS $T_0 \oplus T_1$ is a *conservative operational extension* to TSS $T_0$. Often one wants to add new operators and rules to a given transition system specification (TSS). An (operational) conservative extension requires that an original TSS and its extension prove exactly the same closed transition rules that have only negative premises and an original closed term as their source [AFV99]. In this thesis, the new operators are the fault actions considered for an RTN.

Our idea is to build on the theoretical TTS layer of our SOS semantics for RTNs and then use the results from Aceto et al [AFV99] to show our additional RTN-SL language features, $T_1$ a conservative extension to $T_0$. This result is necessary to show the properties proved of an RTN before the consideration of faults are true in our extended TSS under the assumption that no faults occur. From this, we can then set about showing the fault tolerant mechanism we propose do indeed tolerate the faults considered.

### 5.3.1 Definitions

We begin by defining the notion of *source-dependency* of variables [FV98] which is an important ingredient of a rule format to ensure that an extension of a TSS is operationally conservative. In order to conclude that an extended TSS is operationally conservative over the original TSS, we need to know that the variables in the original transition rules are source-dependent. In the literature this requirement is sometimes overlooked.

**Definition 5.8 (Source-dependency)** *The source-dependent variables in a transition rule* $\rho$ *are defined inductively as follows:*

- *all variables in the source of* $\rho$ *are source-dependent;*
- *if* $t \xrightarrow{a} t'$ *is a premise of* $\rho$ *and all variables in* $t$ *are source-dependent, then all variables in* $t'$ *are source-dependent.*

*A transition rule is source-dependent if all its variables are.*

Source-dependency is a more liberal formulation of the syntactic criterion "pure and well-formed" for formal variables in formal rules. From [FV98] it states:

- that rules have either 'formal' or 'actual' variables
- formal variables are found from $FV(t*)$

**Definition 5.9 (Formal Variables)** $FV(t*)$ *denotes the set of formal variables that occur in the formal term* $t*$

**Definition 5.10 (Sum of TSSs)** *Let* $T_0$ *and* $T_1$ *be TSSs whose signatures* $\Sigma_0$ *and* $\Sigma_1$ *agree on the arity of the function symbols in their intersection. We write* $T_0 \oplus T_1$ *for the union of* $\Sigma_0$ *and* $\Sigma_1$.

*The sum of* $T_0$ *and* $T_1$, *notation* $T_0 \oplus T_1$, *is the TSS over signatures* $\Sigma_0 \oplus \Sigma_1$ *containing the rules in* $T_0$ *and* $T_1$.

We now state the main theorem and show for an example that its semantics are first source-dependent and then that the fault semantics are a conservative extension to $T_0$.

**Theorem 5.1 (Cons.Extn)** *Let* $T_0$ *and* $T_1$ *be TSSs over signature* $\Sigma_0$ *and* $\Sigma_1$ *respectively. Under the following conditions,* $T_0 \oplus T_1$ *is an operational conservative extension of* $T_0$.

1. *Each* $\rho \in T_0$ *is source-dependent.*
2. *For each* $\rho \in T_1$,
   - *either the source of* $\rho$ *is fresh,*
   - *or* $\rho$ *has a premise of the form* $t \xrightarrow{a} t'$ *or* $tP$, *where:*
     - $t \in \mathbb{T}(\Sigma_0)$;

- *all variables in t occur in the source of* $\rho$;
- $t'$, *a or P is fresh.*

For example, consider the transition rule for *tick*.

**Example 5.1** *We consider the* 'Term'*-rule which was introduced in Section 5.1.2, which sets the status of an activity to* TERM *should it be in a state with no successors. This prevents* false *faults being observed, such as a* late *or* crash *fault.*

$$\boxed{term}\ \frac{a \in dom\,\sigma_1 \quad is_(\sigma_1(a), act) \quad \nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(\sigma_1(a).cs, \sigma_1(a).ls, \mathrm{TERM})\}, \{\})}$$

*Since the source of 'tick' is* $(RTN, \Sigma, \Pi)$ *and* $(RTN, \Sigma, \Pi)$ *contains only formal variables, it follows* $(RTN, \Sigma, \Pi)$ *is source-dependent (defn 3.16(1) [FV98]). Since the term-rule has hypotheses*

$$\nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs$$

*and*

$$a \in dom\,\sigma_1$$

*where*

$$\mathrm{FV}(\nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs) = \{RTN, \sigma_1, \pi_1\}$$

*and*

$$\mathrm{FV}(a \in dom\,\sigma_1) = \{\pi_1\}$$

*it follows that each premise is source-dependent (defn 3.16(3) [FV98])*

Similarly, each rule in $T_0$ (in Section 5.1.2) follows the same rule format and structure and has been found to be *source-dependent*, verified by the LETOS tool, described later in Section 5.3.3

### 5.3.2 Faults are a conservative extension

Given the TSS $T_0$ is source-dependent and $T_1$ (as defined previously), we examine whether $T_0 \oplus T_1$ is a conservative extension of $T_0$.

**Corollary 5.1** *Given* $T_0$ *and* $T_1$ *as specified previously,* $T_0 \oplus T_1$ *is a conservative extension of* $T_0$.

PROOF We are required to show that each rule is source-dependent and each new rule in $T_1$ has a premise which is *fresh*.

Each rule in $T_0$ and $T_1$ is source-dependent by inspection, and has been verified by the LETOS tool.

For each rule in $T_1$ which has conclusion of the $\xrightarrow{s}$ relation, it has a premise defined by the fresh relation $\xrightarrow{f}$. That is to say, to make the transition specified for $\xrightarrow{s}$ in $T_1$ new behaviours not specified in $T_0$ must have occurred. Therefore, the rules in $T_1$ prove to be a conservative extension.

■

### 5.3.3 Animating SOS rules with LETOS

The methodology of LETOS is to provide a lightweight tool that offers the most important facilities - type checking, LaTeXscripting, animation and a derivation tree - at a minimal cost.

Hartel addresses the crucial issue of non-determinism in a convenient/compatible way to our own. Hartel [Har97] states:

> An SOS yields a derivation sequence, whereas a natural semantics delivers a derivation tree. To put the two on a level playing field, it is convenient to add another rule to the SOS specifications. This new relation $\stackrel{4}{\Rightarrow}$ given below computes the *transitive closure* of the relation $\stackrel{3}{\Rightarrow}$ , and selects the final state:
>
> $$((S, state) \rightarrow state)$$
> $$\boxed{\stackrel{4}{\Rightarrow}}\ \frac{\vdash < s_1, s > \stackrel{3}{\Rightarrow} < s'_1, s' > \quad \vdash < s'_1, s' > \stackrel{3}{\Rightarrow} s''}{\vdash < s_1, s > \stackrel{4}{\Rightarrow} s''}$$

One further benefit of the LETOS tool is the feature to check for *source dependency* of a rules premise, the base condition for the conservative extension result we use found true of our semantics.

One key feature of LETOS is the ability for a Miranda program (generated by LETOS) not only to calculate the final state, but also the entire derivation tree, given an initial state. This derivation tree forms a proof in a natural deduction style, for which LETOS will render (in LaTeX).

# Chapter 6

# A soundness argument for the axiomatic semantics

## Contents


6.1 **Objectives** . . . . . . . . . . 119
6.2 **Approach** . . . . . . . . . . 119
6.2.1 An Example . . . . . . . . . . 120
6.2.2 Correctness Specification . . . . . . . . . . 122
6.2.3 An informal argument . . . . . . . . . . 124
6.3 **Trace Induction** . . . . . . . . . . 128
6.3.1 Model Space . . . . . . . . . . 128
6.3.2 Induction Rules . . . . . . . . . . 130
6.4 **RTN Soundness argument** . . . . . . . . . . 133
6.4.1 Ax6 . . . . . . . . . . 134
6.5 **Review** . . . . . . . . . . 141


This chapter first gives an introduction to the theory of *inductively defined relations*, of which presentations of operational semantics are examples. This principle of induction, specifically *rule induction*, is used to define the action based set of behaviours for a RTN. An inductive definition consists of introduction rules which define the least closed set of values, or configurations, that satisfy the relations. It specialises to proof rules for reasoning about the operational semantics of RTNs.

We first identify the objectives for formalising a soundness argument which are highlighted through an example. The example demonstrates the necessity of a formal proof and serves to highlight the structure we pursue. We then outline a method to formalise our approach in Isabelle/HOL [NPW02].

## 6.1 Objectives

Regarding semantics, there are three important questions we must face:

1. How can we say our semantics are sound?
2. How can we guarantee our semantics are complete?
3. How can we say there will not be any redundant axioms among all our rules?

These questions are usually addressed in terms of the concepts of soundness and completeness.

**Soundness:** An axiomatic semantics is called sound if it is consistent with an operational (or denotational) model that has already been formulated

**Completeness:** The completeness of an axiomatic semantics is to check if the axioms are sufficient to specify all the behaviours of a model the axioms define. There may also be too many axioms in a system. Redundant axioms may add new features to our language, but these features actually do not exist in our language originally. Therefore we want to detect and avoid these redundant axioms in our system.

First, and foremost, a semantic model should be sound, so that validation of system designs can proceed with confidence in the results. We wish to undertake our formal reasoning in Chapter 8 using the extended axiomatic semantics and therefore require first to show the axioms sound.

## 6.2 Approach

We suggest an approach based around the derived principle of definitions in HOL - the *inductive definition* of relations [CM92, Nip98]. The key element is a derived rule which allows a relation, in our case the transition rules, be defined by giving a set of introduction rules for generating its elements. Structural induction is often inadequate to prove properties of operational semantics, often it is useful to do induction on derivations. A derivation takes the form of a tree, which includes derivations to the premises of a rule instance. Rule instances are got from rules by substituting actual terms or values for meta-variables (or formal variables) in them.

Our operational semantics define a trace model for RTNs, with which we now require to show the axioms sound. The form of a trace is got from the SOS rules, such that properties of a derivation –or trace– can be inductively reasoned about. The properties we wish to prove are the interpretations of the RTN axioms in the trace model, not just for a trace but, for all traces reachable which are specified by the SOS rules.

### 6.2.1 An Example

Because the transition system from the structured operational semantics (SOS) is given by rules, we have an elementary, but very useful, proof technique for proving properties of the operational semantics of RTNs. The simple proof of the equivalence of the axiomatic semantics with its operational unfolding exhibits an important technique: in order to prove consistency it is helpful to consider the various possible forms of derivations [Win93]. This approach is illustrated in Section 6.2.3, but the idea is used again and

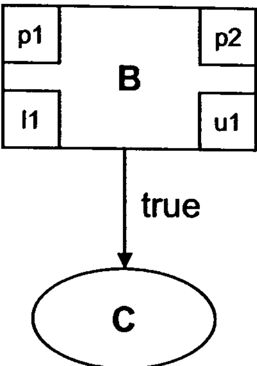



Figure 6.1: Example fragment of an RTN-SL Design

again, though never in such detail. Later we shall show how *rule induction*, in principle, can formalise the technique used here.

Consider now, as an example, the exit transition from a dynamic state. Figure 6.1 shows the arrangement of a dynamic state, $B$ which has an exit transition labelled **true** that transitions to static state $C$. The design also stipulates that the exit transition must occur not earlier than $l1$ time units after entering state $B$ nor later than $u1$ time units.

We wish to show the axioms (given below) generated from this fragment via $\Omega$ are sound with respect to our operational model using our elementary proof technique. This requires we show the existence of a derivation for which the axioms hold, such that the relationship of events specified in an axiom does exist in an operational sense. Section 6.2.1.1 illustrates the axiomatic semantics which describe this example, while Section 6.2.1.2 shows the operational counterparts.

#### 6.2.1.1 Axiomatic Specification

The axiomatic model, from [PAH00], of our example is described by the following axiomatic specifications:

**A1ax3:** $\forall i,j : Occ, t : Time \cdot (\Theta(\downarrow B, i, t) \wedge \Phi(\mathbf{true}, j, t)) \;\Rightarrow\; \Theta(\uparrow C, i, t)$

**A1ax5b:** $\forall i : Occ, t : Time \cdot \Theta(\uparrow C, i, t) \;\Rightarrow\; \Theta(\downarrow B, i, t)$

**A1ax6:** $\forall i,j : Occ, t : Time \cdot \Theta(\uparrow B, i, t) \;\Rightarrow\; \exists t_1 : Time \cdot t + l_1 \leq t_1 \leq t + u_1 \wedge \Theta(\downarrow B, i, t_1)$

#### 6.2.1.2 Operational Model

The operational model defined over the relation $(rtn_1, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$ (which was described in Section 5.2) is the transition from a configuration containing the static specification of our system ($rtn_1$), the current state -or dynamic- properties ($\sigma_1$) and the history of events, a trace ($\pi_1$). To use our elementary proof technique upon our example, we must define (partially) the starting configuration:

$$(rtn, \sigma_1, \pi_1) = \left( \begin{array}{l} \{a1 \rightarrow mk_Activity(\{p1\}, \{p2\}, \text{-}, \{op1\}, \\ \qquad\qquad\qquad\qquad mk_SM(\{B, C\}, \{mk_Transition(B, C, true)\}, \text{-}))\}, \\ \{p1 \rightarrow v1, p2 \rightarrow v2, B \rightarrow mk_dySt([op1])\}, \\ [\{\ \Uparrow B\ \}] \end{array} \right)$$

The relation $\xrightarrow{rtn}$ specifies the evaluation of all possible actions from some configuration to generate the next, where $\xrightarrow{s}$ specifies the evaluation of a single action. We therefore wish to find a sequence, or derivation, of actions (i.e. $\xrightarrow{s}$steps) to show our axioms **can** be true. We leave it until later to show soundness, which requires there does not exist a derivation showing our axiom false, by considering all derivations specified under the $\xrightarrow{rtn}$ relation. The rules which are of interest in showing axiom $A1ax6$ true, and transform this starting configuration, are shown below:

$$\boxed{\text{dySt}}\ \dfrac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops = \{\,\} \\ uact = mk_act(cs, ls, \mathrm{WAIT_TR}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\,\})}$$

$$\boxed{\text{tr-dySt}}\ \dfrac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\ tr \in RTN(a).sm.ts \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ mk_Dynamic_State(id,, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\, \pi_1 \cdot \Uparrow cs \in_i \pi_1(t_1) \wedge t_1 + bcet \leq \mathsf{len}\, \pi_1 < t_1 + wcet \\ \qquad\qquad \nexists t_2 \in \mathsf{inds}\, \pi_1 \cdot t_2 \geq t_1 \wedge \Uparrow cs \in_{i+1} \pi_1(t_2) \\ ins(\sigma_1(tr.trg)) = \mathrm{INDS} \\ uact = mk_act(tr.trg, ls, \mathrm{INDS}) \\ trgs = mk_dySt([op \mid op \in s.ops \cdot s \in RTN(a).sm.ss \wedge s.id = tr.trg]) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, tr.trg \mapsto trgs\}, \{\ \Downarrow tr.src, \Uparrow tr.trg\})}$$

$$\boxed{\text{expr-tr}}\ \dfrac{\begin{array}{l} is_(l, Expression) \\ (l, \sigma_1) \xrightarrow{e} \mathbf{true} \end{array}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true}}$$

### 6.2.2 Correctness Specification

We are interested in axiomatic specifications, specified in RTL, which often take the following forms:

$$\forall i : Occ, t_1 : Time \cdot \Theta(e_1, i, t_1) \ \Rightarrow\ \exists j : Occ, t_2 : Time \cdot \Theta(e_2, j, t_2) \wedge \mathrm{P}(t_1, t_2)$$

and

$$\forall i : Occ, t : Time \cdot \Theta(e_1, i, t) \wedge \Phi(c, i, t) \ \Rightarrow\ \Theta(e_2, i, t)$$

Given the defined relations $\xrightarrow{rtn}$ & $\xrightarrow{s}$ (described in Section 5.2), we seek to represent the equivalent axiomatic specification over our operational semantics. We have discussed in Section C.5.1 the evaluation of

this relation and the set of permutations it generates. We therefore require an interpretation of relating a *typical* axiomatic specification onto our SOS model on which to define our propositions.

We have seen, in Section 4.1, the definition of the RTL event model and the syntax of its formulas. We now define a semantic function, $\mathcal{M}$ that defines the value of RTL formulas ($f_{RTL}$) in terms of our SOS (trace) semantics. The semantic function, $\mathcal{M}$ performs case analysis on the components of the concrete syntax and defines the meaning of RTL formulas thus:

$$\mathcal{M} : f_{RTL} \rightarrow f_{SOS} \rightarrow \mathbb{B}$$

An auxiliary function which makes the semantics more presentable is $\in_i$ which states that an event is occurring for the *ith* time in a trace:

$$e \in_i \pi(t) \triangleq \mathsf{card}\,\{t \in \mathsf{inds}\,\pi \cdot e \in \pi(t)\} = i$$

---

The Semantic Function $\mathcal{M}$

---

$$\begin{aligned}
\mathcal{M}\,[\![P_1 \wedge P_2]\!]_{\sigma,\pi} &\triangleq \mathcal{M}\,[\![P_1]\!]_{\sigma,\pi} \wedge \mathcal{M}\,[\![P_2]\!]_{\sigma,\pi} \\
\mathcal{M}\,[\![P_1 \vee P_2]\!]_{\sigma,\pi} &\triangleq \mathcal{M}\,[\![P_1]\!]_{\sigma,\pi} \vee \mathcal{M}\,[\![P_2]\!]_{\sigma,\pi} \\
\mathcal{M}\,[\![P_1 \Rightarrow P_2]\!]_{\sigma,\pi} &\triangleq \mathcal{M}\,[\![P_1]\!]_{\sigma,\pi} \Rightarrow \mathcal{M}\,[\![P_2]\!]_{\sigma,\pi} \\
\mathcal{M}\,[\![\neg P]\!]_{\sigma,\pi} &\triangleq \neg\mathcal{M}\,[\![P]\!]_{\sigma,\pi} \\
\mathcal{M}\,[\![\exists i : Occ, t : Time \cdot P(i,t)]\!]_{\sigma,\pi} &\triangleq \exists i' : Occ, t' \in \mathsf{inds}\,\pi \cdot M[\![P]\!]_{\sigma,\pi}(i',t') \\
\mathcal{M}\,[\![\forall i : Occ, t : Time \cdot P]\!]_{\sigma,\pi} &\triangleq \forall i' : Occ, t' \in \mathsf{inds}\,\pi \cdot \mathcal{M}\,[\![P]\!]_{\sigma,\pi}(i',t') \\
\mathcal{M}\,[\![\Theta(e,i,t)]\!]_{\sigma,\pi} &\triangleq \mathsf{if}\ t \in \mathsf{inds}\,\pi\ \mathsf{then}\ e \in_i \pi(t)\ \mathsf{else\ false}
\end{aligned}$$

---

This is intended to capture the interpretation of RTL formulas in our operational model. One careful consideration was with respect the $\mathcal{M}\,[\![\neg P]\!]_{\sigma,\pi}$ definition. We wished it be $\neg\mathcal{M}\,[\![P]\!]_{\sigma,\pi}$ which drove out a careful construction of the *guard* of $\mathcal{M}\,[\![\Theta(e,i,t)]\!]_{\sigma,\pi}$.

Consider as an example a typical RTL theorem:

$$Q \equiv \forall i : Occ, t_1 : Time \cdot \Theta(e_1,i,t_1) \Rightarrow \exists j : Occ, t_2 : Time \cdot t_2 \leq t_1 + u \wedge \Theta(e_2,j,t_2)$$

then

$$\begin{aligned}
\mathcal{M}\,[\![Q]\!]_{\sigma,\pi} &\triangleq \mathcal{M}\,[\![\forall i : Occ, t_1 : Time \cdot \Theta(e_1,i,t_1) \Rightarrow \exists j : Occ, t_2 : Time \cdot t_2 \leq t_1 + u \wedge \Theta(e_2,j,t_2)]\!]_{\sigma,\pi} \\
&\triangleq \forall i : Occ, t_1 \in \mathsf{inds}\,\pi \cdot \mathcal{M}\,[\![\Theta(e_1,i,t_1) \Rightarrow \exists j : Occ, t_2 : Time \cdot t_2 \leq t_1 + u \wedge \Theta(e_2,j,t_2)]\!]_{\sigma,\pi} \\
&\triangleq \forall i : Occ, t_1 \in \mathsf{inds}\,\pi \cdot \mathcal{M}\,[\![\Theta(e_1,i,t_1)]\!]_{\sigma,\pi} \Rightarrow \mathcal{M}\,[\![\exists j : Occ, t_2 : Time \cdot t_2 \leq t_1 + u \wedge \Theta(e_2,j,t_2)]\!]_{\sigma,\pi} \\
&\triangleq \forall i : Occ, t_1 \in \mathsf{inds}\,\pi \cdot \mathsf{if}\ t_1 \in \mathsf{inds}\,\pi\ \mathsf{then}\ e_1 \in_i \pi(t_1)\ \mathsf{else\ false} \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot M[\![t_2 \leq t_1 + u \wedge \Theta(e_2,j,t_2)]\!]_{\sigma,\pi} \\
&\triangleq \forall i : Occ, t_1 \in \mathsf{inds}\,\pi \cdot e \in_i \pi(t) \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot t_2 \leq t_1 + u \wedge \mathsf{if}\ t_2 \in \mathsf{inds}\,\pi\ \mathsf{then}\ e_2 \in_j \pi(t_2)\ \mathsf{else\ false} \\
&\triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot e_1 \in_i \pi(t_1) \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot t_2 \leq t_1 + u \wedge e_2 \in_j \pi(t_2)
\end{aligned}$$

which we believe is a correct translation to the conjecture in the SOS model as $\mathcal{M}\,[\![f_{RTL}]\!]_{\sigma,\pi} \in \mathbb{B}$:

$$\forall t_1 \in \mathsf{inds}\,\pi \cdot e_1 \in_i \pi(t_1) \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot t_2 \leq t_1 + u \wedge e_2 \in_j \pi(t_2)$$

### 6.2.3 An informal argument

Consider the problem of evaluating[1] the *tr-dySt* rule in some configuration $(rtn, \sigma_1, \pi_1)$. This amounts to finding a derivation which concludes to the source of the *tr-dySt* rule therefore satisfying its premises.

The search for a derivation is best achieved in a upwards fashion: Start by finding a rule with a conclusion matching the premise of the *tr-dySt* rule; if this is an axiom the derivation is complete; otherwise try to build derivations up from the premises until successful. In general, more than one rule has a source that matches a given configuration. To guarantee finding a derivation tree, all paths must be considered. All possible derivations with conclusions of the correct form should be constructed *in parallel.*

We now show one possible derivation of the operational semantics which shows the axiom ($A1ax6$) sound with respect to the operational model.

**Proposition 6.1** *From our previous example, let*

$$\textit{A1ax6:}\ \forall i : Occ, t_1 : Time \cdot \Theta(\uparrow B, i, t_1) \ \Rightarrow\ \exists t_2 : Time \cdot t_1 + l_1 \leq t_2 \leq t_1 + u_1 \wedge \Theta(\downarrow B, i, t_2)$$

*be sound with respect the configuration* $(rtn, \sigma_1, \pi_1)$.

PROOF We want to show

$$\begin{aligned} \text{A1ax6} \ &\triangleq\ \mathcal{M}[\![\forall i : Occ, t_1 : Time \cdot \Theta(\uparrow B, i, t_1) \ \Rightarrow\ \exists t_2 : Time \cdot t_1 + l_1 \leq t_2 \leq t_1 + u_1 \wedge \Theta(\downarrow B, i, t_2)]\!]_{\sigma,\pi} \\ &\triangleq\ \forall t_1 \in \mathsf{inds}\,\pi \cdot \uparrow B \in_i \pi(t_1) \ \Rightarrow\ \exists t_2 \in \mathsf{inds}\,\pi \cdot t_1 + l_1 \leq t_2 \leq t_1 + u_1 \wedge \downarrow B \in_i \pi(t_2) \end{aligned}$$

is sound and consistent with the configuration $(rtn, \sigma_1, \pi_1)$ and all derivations there from, i.e. $(rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$.

To assert our conjecture for all $\pi$, we are required to show the conjecture holds for all possible formations of $\pi$. That is, to show for each step of the construction of $\pi$, our conjecture is true. In our work, the next value of $\pi$ is defined by the $\xrightarrow{rtn}$ relation, such that $(rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$. Subsequently, for each new $\pi_2$ by $\xrightarrow{rtn}$, we show its formation with respect $\xrightarrow{s}$. We are therefore required to show that the proposition is true first for the $(rtn, \sigma_1, \pi_1)$ configuration -our induction hypothesis- and for the induction step. We argue first that there exists a derivation from the configuration $(rtn, \sigma_1, \pi_1)$ to some $\pi_2$ which shows $A1ax6$ true. Each possible value of $\pi_2$ is specified as a transition (i.e. the next step) under $\xrightarrow{s}$ where $\pi_2$ is the concatenation[2] of $\pi_1$ and $es$ for which $A1ax6$ remains true.

Suppose $A1ax6$ is true for the transition $(rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$ for traces $\pi_1, \pi_2$, then there must exist a derivation of $(rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$. Inspecting the SOS rules for a possible sequence of transition steps, we find that the final rule of the derivation must be the *tr-stSt* rule:

[1] We stated earlier, this informal discussion considers the sequence of actions under the $\xrightarrow{s}$ relation, and not the complete evaluation of a time instance, as specified by $\xrightarrow{rtn}$.

[2] In the derivation sequence, the $\xrightarrow{rtn}$ transitions aren't shown. Rather, between each $\xrightarrow{s}$ transition there would exist the evaluation of the $\xrightarrow{rtn}$ relation (c.f. Section 5.2) to ramify the small-step transitions.

$$\frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 = mk_act(B, ls, \text{WAIT_TR}) \\ \ldots \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \Uparrow B \in_i \pi_1(t_1) \wedge \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot t_2 \geq t_1 \wedge \Uparrow B \in_{i+1} \pi_1(t_2) \wedge \\ \qquad t_1 + bcet \leq \mathsf{len}\,\pi_1 < t_1 + wcet \\ \ldots \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(tr.trg, ls, \text{WAIT_TR})\}, \{\Downarrow B, \Uparrow C\})}$$

In this case, we consider the derivation follows from the *dySt* rule and has the form[3]:

$$\frac{\begin{array}{l} \dfrac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 = mk_act(B, ls, \text{INDS}) \\ \sigma_1(B).ops = [\,] \end{array}}{(rtn_1, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(B, ls, \text{WAIT_TR})\}, \{\,\})} \\ a \in \mathsf{dom}\,\sigma_1 = mk_act(cs, ls, \text{WAIT_TR}) \\ \ldots \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \Uparrow B \in_i \pi_1(t_1) \wedge \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot t_2 \geq t_1 \wedge \Uparrow B \in_{i+1} \pi_1(t_2) \wedge \\ \qquad t_1 + l1 \leq \mathsf{len}\,\pi_1 < t_1 + u1 \\ \ldots \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(tr.trg, ls, \text{WAIT_TR})\}, \{\Downarrow tr.src, \Uparrow tr.trg\})}$$

We can continue the derivation to see we previously must have evaluated the operation specified, which includes a *read* action, we therefore add first the *dySt-op-rd* rule which must follow the *tr-dySt* rule which states we entered state $B$ in the first place:

[3] Double horizontal lines indicate the conclusion of rules within the derivation, as well as the final conclusion concluding our axiom is sound.

$$\dfrac{\dfrac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 \\ \ldots \\ uact = mk_act(B, ls, \mathrm{INDS}) \\ trgs = mk_dySt([op \mid op \in s.ops \cdot s \in RTN(a).sm.ss \wedge s.id = B]) \end{array}}{(rtn_1, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, B \mapsto trgs\}, \{\Downarrow tr.src, \Uparrow B\})}}{}$$

$$\dfrac{\begin{array}{l} \ldots \\ (p1, \sigma_1) \xrightarrow{rd} v \\ ls' = ls \dagger \{p1 \mapsto v\} \\ (post, ls') \xrightarrow{e} \mathbf{true} \end{array}}{(rtn_1, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{B \mapsto \mathsf{tl}\, \sigma_1(B).ops, a \mapsto mk_act(B, ls', \mathrm{INDS})\}, \{rd_p1\})}$$

$$\dfrac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 = mk_act(B, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops = [\,] \end{array}}{(rtn_1, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(B, ls, \mathrm{WAIT_TR})\}, \{\})}$$

$$\dfrac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 = mk_act(B, ls, \mathrm{WAIT_TR}) \\ \ldots \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ \exists t_1 \in \mathsf{inds}\, \pi_1 \cdot \Uparrow B \in_i \pi_1(t_1) \wedge \nexists t_2 \in \mathsf{inds}\, \pi_1 \cdot t_2 \geq t_1 \wedge \Uparrow B \in_{i+1} \pi_1(t_2) \wedge \\ \qquad t_1 + l1 \leq \mathsf{len}\, \pi_1 < t_1 + u1 \\ \ldots \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(C, ls, \mathrm{WAIT_TR})\}, \{\Downarrow B, \Uparrow C\})}$$

Noting the premise that all operations specified on $B$ are completed must indicate we have entered $B$ ($\Uparrow B$) previously and specified the operations to execute on it. Therefore we continue the derivation to find in some previous trace we entered $B$ ($\Uparrow B$) noting the *tr-stSt* rules conclusion matches our upper-most premise to see:

Thus

$$A1ax6 \triangleq \forall t_1 \in \mathsf{inds}\, \pi \cdot \Uparrow B \in_i \pi(t_1) \;\Rightarrow\; \exists t_2 \in \mathsf{inds}\, \pi \cdot t_1 + l_1 \leq t_2 \leq t_1 + u_1 \wedge \Downarrow B \in_i \pi(t_2)$$

holds for $\pi_1, \pi_2$ such that $(rtn_1, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_2)$. ∎

Although we have shown the existence of a derivation that suggests our axiom sound, it does not do so conclusively. Rather, we must consider all possible derivation trees and therefore the exclusion that another derivation path showing the axiom inconsistent with the operational semantic. We therefore require a more formal argument, which appeals to this approach but is formally sound.

## 6.3 Trace Induction

We have elaborated previously (c.f. Section 5.2) that the $\xrightarrow{s}$ relation specifies a *small step* in an RTN, for which many such steps can fire from some configuration. The history -or trace- of an RTN is got from the $\xrightarrow{rtn}$ relation which specifies the transition from configuration to configuration which ramifies each small step to yield a new configuration. Therefore, the trace is inductively defined, such that:

$t_1 : (rtn, \sigma_0, []) \xrightarrow{rtn} (\sigma_1, [es])$
$t_2 : (rtn, \sigma_1, [es]) \xrightarrow{rtn} (\sigma_2, [es] \frown [es'])$
$t_3 : (rtn, \sigma_2, [es] \frown [es']) \xrightarrow{rtn} (\sigma_3, [es, es'] \frown [es''])$

which suggest we can inductively reason that our axiomatic semantics for RTNs are sound over the length of a trace, $\pi$.

### 6.3.1 Model Space

Recall from Chapter 5, the model space of our operational semantics is that of a trace structure: specifically those RTN traces which are reachable by the SOS rules. The formal presentation below is derived from standard sequence (or list) and set induction principles. We discuss these rules, their proofs and the model space we consider for giving a rigorous argument to the soundness of the axiomatic semantics with respect to our SOS rules.

As a reminder, the data types we are interested in are as follows. $\Pi$ is the type of traces, where each trace is a sequence of sets of events, each set representing the events taking place at a time.

$Event = Token$
$ES = Event\text{-set}$
$\Pi = ES^*$

Rather than reason in this model space, which requires repetition of work to argue the traces are valid RTN traces, we prefer to develop a model theory for RTN traces in their own rights. The traces which are valid for a given RTN form a subtype of $\Pi$. We will therefore need inference rules that support reasoning over this subtype.

A particularly useful strategy when considering a theory describing some subtype of a type which as an associated induction rule is to develop a specialisation of the induction rule for that subtype. In general, formulating this rule amounts to putting additional constraints on the induction step (in the form of extra premises in the sequent hypothesis which represents it in the rule) to ensure that it steps from one value of the subtype to the next. Also it is necessary to modify the typing information throughout the rule, and possibly the base case as well if the base case for the main type is not a member of the subtype. For RTN traces, it is easy to see that the induction base case cannot be the empty sequence as this is not a member of the subtype. Rather, these non-empty traces require the property is valid for the singleton sequence (our special case for RTN initialisation). Additionally, it is foreseeable that the induction step can be restricted to step only between RTN traces if an additional constraint is imposed to the effect that the current history is reachable from the history of the last time instance. The restricted data types and their auxiliary definitions are given below.

$Event = Token$

$ES = Event\text{-set}$

$s\text{-}step(rtn : RTN, \sigma : \Sigma, \pi : \Pi_{rtn}, hb : ES)\ r : ES$
post if $(rtn, \sigma, \pi) \overset{s}{\nrightarrow}$ then $r = hb$
else $\exists \sigma' : \Sigma \cdot (rtn, \sigma, \pi) \xrightarrow{s} (\sigma', es) \wedge$
$r = s\text{-}step(rtn, \sigma', \pi, hb \cup es)$

The *s-step* function implicitly builds an *handbag* (*hb*) of events that occur for a single step. From some configuration, *s-step* recursively builds the handbag of events until no further single-steps can fire from the previously generated intermediate configuration $(rtn, \sigma', \pi)$. Note, for each recursive small-step the transition rule $\xrightarrow{s}$ is fired from $\pi$ which enforces out atomicity model.

$init\text{-}rtn(rtn : RTN)\ \sigma : \Sigma$
is not yet specified

$$\Pi_{rtn} = ES^{+}$$
$$\text{inv}\ \pi \frown [s] \triangleq \text{if}\ \pi = [\,]\ \text{then}\ \exists \sigma : \Sigma \cdot (rtn, init\text{-}rtn(rtn), [\,]) \xrightarrow{rtn} (\sigma, [s])$$
$$\text{else}\ \exists \sigma_1, \sigma_2 : \Sigma \cdot s = s\text{-}step(rtn, \sigma_1, \pi, \{\,\}) \wedge (rtn, \sigma_1, \pi) \xrightarrow{rtn} (\sigma_2, \pi \frown [s])$$

The $\Pi_{rtn}$ type invariant above states that each the current trace was reached from a previous RTN trace by some collection of small-steps. This allows one to reason inductively over the subtype.

Our decision not to specify an individual invariant on *ES* is due to the close dependency between *ES* and $\Pi_{rtn}$. Instead, we give only an invariant on $\Pi_{rtn}$ (which is an RTN trace with which we wish to show the axioms sound) and an auxiliary function, *s-step* to restrict the event-sets we concatenate to a RTN trace. Binding the invariant to a data type was a design decision made in VDM in the 1980's, and which separates it from Z [Hay93, WD96]. Previously the invariant was an auxiliary function separate from the type definition. However, the tight binding between the data type definition and its invariant allows the introduction of the invariant predicate into a proof at the necessary stage. Specifically with our model theory, the invariant on $\Pi_{rtn}$ states the step between values is defined by the $\xrightarrow{rtn}$ relation, which in turn states that each small step is defined by the $\xrightarrow{s}$ relation, allowing a proof to only be concerned by valid traces. This allows each proof to extract the properties we wish to show are sound.

### 6.3.2 Induction Rules

The traces which are valid for a given RTN form a subtype of $\Pi$. We will therefore need inference rules that support reasoning over this subtype. We first consider the induction rule for non-empty sequences:

$$\boxed{A^{+}\text{-indn}}\ \frac{s : A^{+} \quad a : A \vdash P([a]) \quad a : A, s' : A^{+}, P(s') \vdash P(s' \frown [a])}{P(s)}$$

Considering now a restricted type of $A^{+}$ we specify

$$B = A^{+}$$
$$\text{inv}\ b \triangleq I(b)$$

we derive our induction rule to be:

$$\boxed{\text{B-indn}}\ \frac{b : A^{+} \quad a : A \vdash I([a]) \Rightarrow P([a]) \quad a : A, s' : A^{+}, I(s') \Rightarrow P(s') \vdash I(s' \frown [a]) \Rightarrow P(s' \frown [a])}{I(b) \Rightarrow P(b)}$$

or using inv-$b : A^+ \to \mathbb{B}$

$$\boxed{\text{B-indn}}\ \frac{b : A^+ \quad a : A \vdash \text{inv-}B([a]) \Rightarrow P([a]) \quad a : Z, s' : A^+, \text{inv-}B(s') \Rightarrow P(s') \vdash \text{inv-}B(s' \frown [a]) \Rightarrow P(s' \frown [a])}{\text{inv-}B(b) \Rightarrow P(b}$$

We now apply the same derivation to our data types and derive a rule for non-empty sequence of event sets:

$$\boxed{\Pi_{rtn}\text{-ind1}}\ \frac{\pi : ES^+ \quad es : ES \vdash \text{inv-}\Pi_{rtn}([es]) \Rightarrow P([es]) \quad es : ES, \pi' : ES^+, \text{inv-}\Pi_{rtn}(\pi') \Rightarrow P(\pi') \vdash \text{inv-}\Pi_{rtn}(\pi' \frown [es]) \Rightarrow P(\pi' \frown [es])}{\text{inv-}\Pi_{rtn}(\pi) \Rightarrow P(\pi)}$$

and prove it trivially

PROOF ($\Pi_{rtn}$-IND1)

from $\pi : ES^+$
  $es : ES \vdash \text{inv-}\Pi_{rtn}([es]) \Rightarrow P([es])$
  $es : ES, \pi' : ES^+, \text{inv-}\Pi_{rtn}(\pi') \Rightarrow P(\pi') \vdash \text{inv-}\Pi_{rtn}(\pi' \frown [es]) \Rightarrow P(\pi' \frown [es])$
1  from $es : ES$
   infer $\text{inv-}\Pi_{rtn}(es) \Rightarrow P([es])$ — sequent(h2,1.h1)
2  from $es : ES, \pi' : ES^+, \text{inv-}\Pi_{rtn}(\pi') \Rightarrow P(\pi')$
   infer $\text{inv-}\Pi_{rtn}(\pi' \frown [es]) \Rightarrow P(\pi' \frown [es])$ — sequent(h3,2.h1,2.h2,2.h3)
infer $\text{inv-}\Pi_{rtn}(\pi) \Rightarrow P(\pi)$ — $A^+$-ind(1,2)

■

The first rule below is our induction rule over the restricted type: RTN Traces, $\Pi_{rtn}$} and the second is the induction rule over event sets that arise at each point in the trace.

$$\boxed{\Pi_{rtn}\text{-indn}}\ \frac{\pi : \Pi_{rtn}; \quad s : ES,\ \mathit{inv}\text{-}\Pi_{rtn}([s]) \vdash_s P([s]); \quad es : ES,\ \pi' : \Pi_{rtn},\ P(\pi'),\ \mathit{inv}\text{-}\Pi_{rtn}(\pi' \frown [es]) \vdash_{\pi',es} P(\pi' \frown [es])}{P(\pi)}$$

$$\boxed{ES\text{-indn}}\ \frac{\pi : \Pi_{rtn}; \quad s : ES, P(\pi), s = \{\} \vdash P(\pi \frown [\{\}]) \quad es : ES,\ s' : ES,\ P(\pi \frown [s']),\ \mathit{inv}\text{-}\Pi_{rtn}(\pi \frown [s']),\ s' \cap es = \{\} \vdash_{es} P(\pi \frown [s' \cup es)])}{P(\pi \frown [s])}$$

The proof below makes use of the previous inference rules for trace induction:

PROOF ($\Pi_{rtn}$-INDN)

from $\pi : \Pi_{rtn}$;
$\quad s : ES,\ inv\text{-}\Pi_{rtn}([s]) \vdash_s P([s])$;
$\quad es : ES,\ \pi' : \Pi_{rtn},\ P(\pi'),\ inv\text{-}\Pi_{rtn}(\pi' \frown [es]) \vdash_{\pi', es} P(\pi' \frown [es])$

| | | |
|---|---|---|
| 1 | $\Pi : ES^+$ | h1, $\Pi_{rtn}$ |
| 2 | from $x : ES$ | |
| 2.1 | from inv-$\Pi_{rtn}([x])$ | |
| | infer $P([x])$ | sequent h2(2.h1,2.h1,2.1.h1) |
| | infer inv-$\Pi_{rtn}([x]) \Rightarrow P([x])$ | $\Rightarrow$-I(2.1) |
| 3 | from $x : ES, \pi_x : ES^+$, inv-$\Pi_{rtn}(\pi_x) \Rightarrow P(\pi_x)$ | |
| 3.1 | from inv-$\Pi_{rtn}(\pi_x \frown [x])$ | |
| 3.1.1 | inv-$\Pi_{rtn}(\pi_x)$ | Lemma1 |
| 3.1.2 | $\pi_x : \Pi_{rtn}$ | $\Pi_{rtn}$,3.h2,3.1.1 |
| 3.1.3 | $P(\pi_x)$ | $\Rightarrow$-E(3.h3,3.1.1) |
| | infer $P(\pi_x \frown [x])$ | sequent h3(3.h1,3.1.2,3.1.3,3.1.1) |
| | infer inv-$\Pi_{rtn}(\pi_x \frown [x]) \Rightarrow P(\pi_x \frown [x])$ | $\Rightarrow$-I(3.1) |
| 4 | inv-$\Pi_{rtn}(\pi) \Rightarrow P(\pi)$ | $\Pi_{rtn}$-indn(1,2,3) |
| 5 | inv-$\Pi_{rtn}(\pi)$ | h1,$\Pi_{rtn}$ |
| | infer $P(\pi)$ | $\Rightarrow$-E(5,4) |

■

PROOF (ES-INDN)

from $\pi : \Pi_{rtn}$;
$\quad s : ES, P(\pi), s = \{\} \vdash P(\pi \frown [\{\}])$
$\quad es : ES,\ s' : ES,\ P(\pi \frown [s']),\ inv\text{-}\Pi_{rtn}(\pi \frown [s']),\ s' \cap es = \{\} \vdash_{es} P(\pi \frown [s' \cup es])) \pi : \Pi_{rtn}$

| | | |
|---|---|---|
| 1 | from $s : ES, P(\pi), s = \{\}$ | |
| | infer $P(\pi \frown [\{\}])$ | sequent h2(1.h1,h1,1.h3) |
| 2 | from $\pi_x, es : ES, s' : ES, s' \cap es = \{\}$, inv-$\Pi_{rtn}(\pi_x) \Rightarrow P(\pi_x)$ | |
| 2.1 | from inv-$\Pi_{rtn}(\pi_x \frown [s' \cup es])$ | |
| 2.1.1 | inv-$\Pi_{rtn}(\pi_x \frown [s'])$ | Lemma2(2.1.h1) |
| 2.1.2 | $\pi_x \frown [s'] : \Pi_{rtn}$ | $\Pi_{rtn}$ |
| 2.1.3 | $P(\pi_x \frown [s'])$ | $\Rightarrow$-E(2.h5) |
| | infer $P(\pi_x \frown [s' \cup es])$ | sequent h3(2.h2,2.h3,2.1.3,2.1.1,2.h4) |
| | infer inv-$\Pi_{rtn}(\pi_x \frown [s' \cup es]) \Rightarrow P(\pi_x \frown [s' \cup es])$ | $\Rightarrow$-I(2.1) |
| | infer $P(\pi \frown [s])$ | set-indn(1,2) |

■

The following lemmas where made use of:

$$\boxed{\text{Lemma1}}\ \frac{\begin{array}{c}\pi : ES^{+} \\ es : ES \\ \text{inv-}\Pi_{rtn}(\pi \frown [es])\end{array}}{\text{inv-}\Pi_{rtn}(\pi)}$$

$$\boxed{\text{Lemma2}}\ \frac{\begin{array}{c}\pi : ES^{+} \\ s : ES, es : ES \\ s \cap es = \{\} \\ \text{inv-}\Pi_{rtn}(\pi \frown [s \cup es])\end{array}}{\text{inv-}\Pi_{rtn}(\pi \frown [s])}$$

## 6.4 RTN Soundness argument

Given the axiomatic *schema* for RTNs, we wish to show each axiom sound with respect to our SOS model. However, given the comprehensive presentation of the schema in Chapter 4, we show here only specific examples of this schema sound, specifically those generated by our running example. However, the example specification does explore the more challenging side of the semantics. Therefore, for each axiom Ax1 through Ax13 we propose its interpretation in terms of our SOS model (described in Section 6.2.2) and argue its soundness.

Each axiom is phrased as a conjecture over $\pi : \Pi_{rtn}$ which are the results of each axiom under the semantic (interpretation) function $\mathcal{M}[\![]\!]$, such that $P(\pi)$ must hold. We therefore must show, for each axiom it holds for all $\pi$, i.e. $\forall \pi : \Pi_{rtn} \cdot P(\pi)$. This implicit quantification is left assumed. Not every proof is shown in the body of the thesis, and the reader is referred instead to Appendix B for the complete rigorous proofs, and for the proofs of useful lemmas that we identify and use in the following arguments.

### 6.4.1 Ax6

We now show *Ax6* sound with respect to our SOS model. We repeat the RTL definition of *Ax6* from Section 6.2.1.1 above:

**A1ax6:** $\forall i,j : Occ, t : Time \cdot \Theta(\uparrow B, i, t) \Rightarrow \exists t_1 : Time \cdot t + l_1 \leq t_1 \leq t + u_1 \wedge \Theta(\downarrow B, i, t_1)$

We define –to aid presentation– the axiomatic specification as $P(\pi)$ which is the interpretation of *Ax6* using the $\mathcal{M}[\![]\!]$ semantic function:

$$P(\pi) \triangleq \forall t_1 \in \text{inds}\,\pi \cdot \uparrow s \in_i \pi(t_1) \wedge \text{len}\,\pi \geq t_1 + u \Rightarrow \exists t_2 \in \text{inds}\,\pi \cdot \downarrow s \in_i \pi(t_2) \wedge t_2 \leq t_1 + u$$

Our approach is first to induct over the length of $\pi$, and for each step we can take from some configuration $(rtn, \sigma_1, \pi_1)$ we show the derived trace $(\pi_2)$ satisfies our property $P(\pi_2)$ by trace induction.

### Intuition for Ax6-Soundness :: Proof 6

The form of $P(\pi)$ suggests a structure for a proof. Given that a state entry entry event $(\uparrow s)$ should cause a state event $(\downarrow s)$ within some time of the first, we are required to show that for the case $\uparrow s$ is recorded in

$\pi$, some $\Downarrow s$ event occurs also. The interesting case is when the deadline ($\mathsf{len}\,\pi \curvearrowright [s] = t_1 + u$) is reached for this event. At this deadline ($\mathsf{len}\,\pi \curvearrowright [s] = t_1 + u$), we must show that either the event ($\Downarrow s$) has occurred previously or does so in the final step. This is shown by Proof 7.

PROOF (AX6-SOUNDESS)

| Line | Step | Justification |
|---|---|---|
| | from $rtn : RTN;\ \sigma : \Sigma;\ \pi : \Pi_{rtn};\ t_1, u : \mathbb{N}_1;\ t_1 \in \mathsf{inds}\,\pi$ | |
| 1 | from $es : ES$, $\mathsf{inv}\text{-}\Pi_{rtn}([es])$ | |
| 1.1 | $\mathsf{len}\,[es] = 1$ | |
| 1.2 | $\mathsf{len}\,[es] < t_1 + u$ | h4 |
| 1.3 | $\Uparrow s \in_i [es](t_1) \wedge \mathsf{len}\,[es] \geq t_1 + u \Rightarrow \exists t_2 : Time \cdot \Downarrow s \in_i [es](t_2) \wedge t_2 \leq t_1 + u$ | vac-$\Rightarrow$-I(1.2) |
| | infer $P([es])$ | folding |
| 2 | from $\pi_1 : \Pi_{rtn};\ s : ES; P(\pi_1);\ inv\text{-}\Pi(\pi_1 \curvearrowright [s])$ | |
| 2.1 | $\mathsf{len}\,\pi_1 \curvearrowright [s] < t_1 + u \vee \mathsf{len}\,\pi_1 \curvearrowright [s] > t_1 + u \vee \mathsf{len}\,\pi_1 \curvearrowright [s] = t_1 + u$ | $\vee$-I |
| 2.2 | from $\mathsf{len}\,\pi_1 \curvearrowright [s] < t_1 + u$ | |
| | infer $P(\pi_1 \curvearrowright [s])$ | $\Rightarrow$-I-right-vac(2.2.h1) |
| 2.3 | from $\mathsf{len}\,\pi_1 \curvearrowright [s] > t_1 + u$ | |
| 2.3.1 | from $\Downarrow s \in s$ | |
| 2.3.1.1 | $t_2 > \mathsf{len}\,\pi_1 \curvearrowright [s]$ | |
| 2.3.1.2 | $t_2 > t_1 + u$ | |
| | infer $\neg P(\pi_1 \curvearrowright [s])$ | $\neg$-$\Rightarrow$-I(h5,2.3.1.2) |
| | infer $P(\pi_1 \curvearrowright [s])$ | contradiction(2.3,2.3.1) |
| 2.4 | $(rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_1 \curvearrowright [s])$ | unfolding(2.h3) |
| 2.5 | from $P(\pi_1); (rtn, \sigma_1, \pi_1) \xrightarrow{rtn} (\sigma_2, \pi_1 \curvearrowright [s]);$ $\mathsf{len}\,\pi_1 \curvearrowright [s] = t_1 + u; \Uparrow s \in_i \pi_1(t_1)$ | |
| | infer $P(\pi_1 \curvearrowright [s])$ | Lemma3(2.5) |
| | infer $P(\pi_1 \curvearrowright [s])$ | $\vee$-E(2.1,2.2,2.3,2.5) |
| | infer $\forall \pi : \Pi_{rtn} \cdot P(\pi)$ | $\Pi_{rtn}$-indn(1,2) |

■

## Intuition for Lemma3 :: Proof 7

$$\text{Lemma3}\ \frac{t_1, u : \mathbb{N};\ rtn : RTN;\ \sigma_1, \sigma_2 : \Sigma;\ \pi : \Pi_{rtn};\quad P(\pi);\ s : ES;\ (rtn, \sigma_1, \pi) \xrightarrow{rtn} (\sigma_2, \pi \curvearrowright [s]);\quad \exists t_1 \in \mathsf{inds}\,\pi \cdot \Uparrow s \in_i \pi(t_1) \wedge \mathsf{len}\,\pi \curvearrowright [s] = t_1 + u}{P(\pi \curvearrowright [s])}$$

From the knowns, we must show that the exit state event ($\Downarrow s$) has either already occurred in $\pi_1$ (i.e. by hypothesis), or happens in the next RTN-step. We therefore show, by induction over the next steps that if the exit state events has not occurred, then it does so. Given more than one small-step can occur in each RTN-step, we prove this by induction on $s$.

Given that some *small-steps* have already been taken, recorded in $s$, then we show in the induction step that again, $\Downarrow s$ has either occurred in $s$ or does so in some $\xrightarrow{s}$-step – namely the *tr-stSt* step.

PROOF (LEMMA3)

| | | |
|---|---|---|
| | from $t_1, u : \mathbb{N};\ rtn : RTN;\ \sigma_1, \sigma_2 : \Sigma;\ \pi : \Pi_{rtn};$ | |
| | $\quad P(\pi);\ s : ES;\ (rtn, \sigma_1, \pi) \xrightarrow{rtn} (\sigma_2, \pi \curvearrowright [s]);$ | |
| | $\quad \exists t_1 \in \mathsf{inds}\,\pi \cdot \upharpoonleft s \in_i \pi(t_1) \wedge \mathsf{len}\,\pi \curvearrowright [s] = t_1 + u$ | |
| 1 | $\{\} : ES$ | ES-form |
| 2 | $\exists t_1 \in \mathsf{inds}\,\pi \cdot \upharpoonleft s \in_i \pi(t_1)$ | $\exists$-$\wedge$-E-right(h8) |
| 3 | $\exists t_1 \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \upharpoonleft s \in_i \pi \curvearrowright [\{\}](t_1)$ | preserve(1,h4,h7,2) |
| 4 | from $\exists t_1 \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \upharpoonleft s \in_i \pi \curvearrowright [\{\}](t_1) \wedge \mathsf{len}\,\pi \curvearrowright [\{\}] = t_1 + u$ | |
| 4.1 | $\quad \exists t_2' \in \mathsf{inds}\,\pi \cdot \downharpoonleft s \in_i \pi(t_2')$ | find-$\downharpoonleft s$(h7,h8,1) |
| 4.2 | $\quad \mathsf{len}\,\pi \curvearrowright [\{\}] = \mathsf{succ}(\mathsf{len}\,\pi)$ | len-$\curvearrowright$($\{\}$,h4) |
| 4.3 | $\quad \mathsf{len}\,\pi < \mathsf{len}\,\pi \curvearrowright [\{\}]$ | n<succ(n)(4.2) |
| 4.4 | $\quad t_2' \leq t_1 + u$ | rewriting(4.3,h8) |
| 4.5 | $\quad \exists t_2' \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \downharpoonleft s \in_i \pi \curvearrowright [\{\}](t_2')$ | preserve(1,h4,h7,4.1) |
| | $\quad$ infer $\exists t_2' \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \downharpoonleft s \in_i \pi \curvearrowright [\{\}](t_2') \wedge t_2' \leq t_1 + u$ | $\wedge$-I(4.5,4.4) |
| 5 | $\exists t_1 \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \upharpoonleft s \in_i \pi \curvearrowright [\{\}](t_1) \wedge \mathsf{len}\,\pi \curvearrowright [\{\}] \geq t_1 + u$ $\Rightarrow \exists t_2' \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \downharpoonleft s \in_i \pi \curvearrowright [\{\}](t_2') \wedge t_2' \leq t_1 + u$ | $\Rightarrow$-I(4.h1,4) |
| 7 | $\forall t_1 \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \upharpoonleft s \in_i \pi \curvearrowright [\{\}](t_1) \wedge \mathsf{len}\,\pi \curvearrowright [\{\}] \geq t_1 + u$ $\Rightarrow \exists t_2' \in \mathsf{inds}\,\pi \curvearrowright [\{\}] \cdot \downharpoonleft s \in_i \pi \curvearrowright [\{\}](t_2') \wedge t_2' < t_1 + u$ | $\forall$-I(5) |
| 8 | $P(\pi \curvearrowright [\{\}])$ | folding(7) |
| 9 | from $es : ES;\ s' : ES;\ P(\pi \curvearrowright [s']); inv\text{-}\Pi_{rtn}(\pi \curvearrowright [s']);\ s' \cap es = \{\}$ | |
| | $\quad \exists t_1 \in \mathsf{inds}\,\pi \curvearrowright [s'] \cdot \upharpoonleft s \in_i \pi \curvearrowright [s'](t_1) \wedge \mathsf{len}\,\pi \curvearrowright [s'] = t_1 + u$ | |
| 9.1 | $\quad (rtn, \sigma_1, \pi_1) \xrightarrow{s} (\sigma', es)$ | unfolding(9.h4) |
| 9.2 | $\quad \downharpoonleft s \in es \vee \downharpoonleft s \notin es$ | $\in$-$\vee$-$\notin$ |
| 9.3 | $\quad$ from $\downharpoonleft s \in es$ | |
| 9.3.1 | $\quad\quad \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \upharpoonleft s \in_i \pi_1(t_1)$ | $\exists$-$\wedge$-E-right(h8) |
| 9.3.2 | $\quad\quad \nexists t_2' \in \mathsf{inds}\,\pi_1 \cdot \downharpoonleft s \in_i \pi_1(t_2') \wedge t_2' \geq t_1$ | otherwise $P(\pi_1)$ holds |
| 9.3.3 | $\quad\quad \exists t_1 \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \mathsf{len}\,\pi_1 \curvearrowright [s' \cup es] \leq t_1 + u$ | $\exists$-$\wedge$-E-left(h8) |
| | $\quad\quad$ let $t_2'' = \mathsf{len}\,\pi_1 \curvearrowright [s' \cup es]$ | |
| 9.3.4 | $\quad\quad t_2'' \leq t_1 + u$ | rewriting(9.3.3) |
| 9.3.5 | $\quad\quad \downharpoonleft s \in_i \pi_1 \curvearrowright [s' \cup es](\mathsf{len}\,\pi_1 \curvearrowright [s' \cup es])$ | rtn-step(9.1,9.3.h1) |
| 9.3.6 | $\quad\quad \exists t_2'' \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \downharpoonleft s \in_i \pi_1 [s' \cup es](t_2'')$ | $\exists$-I($t_2''$,9.3.5) |
| | $\quad$ infer $\exists t_2'' \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \downharpoonleft s \in_i \pi_1 [s' \cup es](t_2'') \wedge t_2'' \leq t_1 + u$ | $\wedge$-I(9.3.6,9.3.4) |
| 9.4 | $\quad$ from $\downharpoonleft s \notin es$ | |
| 9.4.1 | $\quad\quad \downharpoonleft s \in s'$ | hypothesis |
| | $\quad$ infer $\exists t_2'' \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \downharpoonleft s \in_i \pi_1 [s' \cup es](t_2'') \wedge t_2'' \leq t_1 + u$ | as 9.3 |
| | infer $\exists t_2'' \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \downharpoonleft s \in_i \pi_1 [s' \cup es](t_2'') \wedge t_2'' \leq t_1 + u$ | $\vee$-E(9.2,9.3,9.4) |
| 10 | $\exists t_1 \in \mathsf{inds}\,\pi \curvearrowright [s'] \cdot \upharpoonleft s \in_i \pi \curvearrowright [s'](t_1) \wedge \mathsf{len}\,\pi \curvearrowright [s'] = t_1 + u \Rightarrow$ $\exists t_2'' \in \mathsf{inds}\,\pi_1 \curvearrowright [s' \cup es] \cdot \downharpoonleft s \in_i \pi_1 [s' \cup es](t_2'') \wedge t_2'' \leq t_1 + u$ | $\Rightarrow$-I(9.h6, 9) |
| 11 | $P(\pi_1 \curvearrowright [s' \cup es])$ | folding(10) |
| | infer $P(\pi_1 \curvearrowright [s])$ | $ES$-indn(8,11) |

∎

The following lemmas have been used.

It is obvious to see the length of a sequence, concatenated with a singleton sequence is the succent of the length of the sequence:

$$\boxed{\text{len-}\curvearrowright}\ \frac{a:A, s:A^*}{\mathsf{len}\, s \curvearrowright [a] = succ(\mathsf{len}\, s)}$$

Assuming RTN steps only ever add events to a trace –which they do– then the occurrence of an event at time, *tO* must occur for the same occurrence at *tO* in the extended sequence.

$$\boxed{\text{preserve-}\curvearrowright}\ \frac{\begin{array}{c} t, t', i:\mathbb{N}_1;\ rtn:RTN;\ \sigma_1, \sigma_2:\Sigma;\ \pi:\Pi_{rtn};\ e:Event;\ s:ES; \\ (rtn, \sigma_1, \pi) \xrightarrow{rtn} (\sigma_2, \pi \curvearrowright [s]) \\ \exists t \in \mathsf{inds}\, \pi \cdot e \in_i \pi(t) \end{array}}{\exists t' \in \mathsf{inds}\, \pi \curvearrowright [s] \cdot e \in_i \pi \curvearrowright [s](t')}$$

The find-$\downarrow\!s$ lemma asserts that if the length of the next RTN step requires an event should occur to meet its deadline, then it must occur in this step. A RTN step is only complete only when all small-step rules that can fire, have done so. Further, the SOS rules state that a state exit event must occur before its deadline. Therefore, having shown our semantics sound, we conclude by contradiction this lemma is true.

$$\boxed{\text{find-}\downarrow\!s}\ \frac{\begin{array}{c} rtn:RTN;\ \sigma_1, \sigma_2:\Sigma;\ \pi:\Pi_{rtn};\ t_1, u, i:\mathbb{N}_1; \\ (rtn, \sigma_1, \pi) \xrightarrow{rtn} (\sigma_2, \pi \curvearrowright [s]); \\ \exists t_1 \in \mathsf{inds}\, \pi \cdot\ \uparrow\!s \in_i \pi(t_1) \wedge \mathsf{len}\, \pi \curvearrowright [s] = t_1 + u;\ \downarrow\!s \notin s \end{array}}{\exists t'_2 \mathsf{inds}\, \pi \cdot\ \downarrow\!s \in_i \pi(t'_2)}$$

The *rtn-step* lemma asserts that if an event occurs in the current step, then it can be found at the end of this trace.This value, $\mathsf{len}\, \pi \curvearrowright [s]$ marks the next time increment from which the event will be visible.

$$\boxed{\text{rtn-step}}\ \frac{\begin{array}{c} rtn:RTN;\ \sigma, \sigma':\Sigma;\ \pi:\Pi_{rtn};\ e:Event;\ es:ES; \\ (rtn, \sigma, \pi) \xrightarrow{s} (\sigma', es); e \in es \end{array}}{e \in \pi \curvearrowright [s](\mathsf{len}\, \pi \curvearrowright [s])}$$

Should there exist a time at which an event occurred, then it can be said there existed a previous configuration from which a *small-step* rule generated the event. This holds true by the definition of the trace data type invariant.

$$\boxed{\text{lemma4}}\ \frac{\begin{array}{c} i, t:\mathbb{N}_1;\ \pi:\Pi_{rtn}; \\ \exists t \in \mathsf{inds}\, \pi \cdot e \in_i \pi(t) \end{array}}{\exists \sigma', \sigma'':\Sigma \cdot (rtn, \sigma', \pi(1, \ldots, t)) \xrightarrow{s} (\sigma'', es) \wedge e \in es}$$

Similarly, should the final associated operations to a dynamic state have been evaluated, then there exists a configuration change recording this state.

$$\boxed{\text{eval-ops}}\ \frac{\begin{array}{c} rtn:RTN;\ \sigma, \sigma':\Sigma;\ \pi:\Pi_{rtn}; \\ (rtn, \sigma, \pi) \xrightarrow{s} (\sigma', es) \wedge \sigma'(a).status = \text{INDS}; \\ \mathsf{let}\ cs = \sigma'(a).cs;\ \sigma'(cs) \neq [] \end{array}}{\exists \sigma'':\Sigma \cdot (rtn, \sigma', \pi \curvearrowright [s \cup es]) \xrightarrow{rtn} (\sigma'', \pi \curvearrowright [s \cup es, \{\}]) \wedge \sigma''(cs) = []}$$

PROOF (FIND-$\downarrow\!s$) As an example, the proof of *find-*$\downarrow\!s$ is shown below:

| | | |
|---|---|---|
| | from $rtn : RTN;\ \sigma_1, \sigma_2 : \Sigma;\ \pi : \Pi_{rtn};\ t_1, u, i : \mathbb{N}_1;$ | |
| | $\quad \exists t_1 \in \mathsf{inds}\, \pi \cdot \upharpoonleft s \in_i \pi(t_1) \wedge \mathsf{len}\, \pi \curvearrowright [\{\}] = t_1 + u$ | |
| 1 | $\exists t_1 \in \mathsf{inds}\, \pi \cdot \upharpoonleft s \in_i \pi(t_1)$ | $\exists$-$\wedge$-E-right(h?) |
| 2 | $\exists \sigma_1', \sigma_1'' : \Sigma, es : ES \cdot (rtn, \sigma_1', \pi(1, \ldots, t_1)) \xrightarrow{s} (\sigma_1'', es) \wedge \upharpoonleft s \in es$ | lemma2(h?) |
| 3 | from $\sigma_1', \sigma_1'' : \Sigma, es : ES;\ (rtn, \sigma_1', \pi(1, \ldots, t_1)) \xrightarrow{s} (\sigma_1'', es) \wedge \upharpoonleft s \in es$ | |
| 3.1 | $(rtn, \sigma_1', \pi(1, \ldots, t_1)) \xrightarrow{s} (\sigma_1'', es)$ | $\wedge$-E-right(3.h2) |
| | let $\pi' = \pi(1, \ldots, t_1)$ | |
| 3.3 | $(rtn, \sigma_1', \pi') \xrightarrow{s} (\sigma_1'', es)$ | rewriting (3.1) |
| 3.4 | $\upharpoonleft s \in es$ | $\wedge$-E-left(3.h2) |
| 3.5 | $\xrightarrow{s}$ :tr-dySt | which-rule(3.3,3.4) |
| 3.6 | $\sigma_1''(a).status = \mathrm{INDS}$ | conc-tr-dySt(3.3,3.5) |
| 3.7 | $\sigma_1''(s) = [\ldots]$ | conc-tr-dySt(3.3,3.5) |
| | infer $(rtn, \sigma_1', \pi') \xrightarrow{s} (\sigma_1'', es) \wedge \sigma_1''(a).status = \mathrm{INDS}$ | $\wedge$-I(3.3,3.6) |
| 4 | $(rtn, \sigma_1', \pi') \xrightarrow{s} (\sigma_1'', es) \wedge \sigma_1''(a).status = \mathrm{INDS}$ | $\exists$-E(2,3) |
| 6 | $\exists \sigma_1''' : \Sigma, s' : ES \cdot (rtn, \sigma_1'', \pi' \curvearrowright [s' \cup es]) \xrightarrow{rtn} (\sigma_1''', \pi' \curvearrowright [s' \cup es, \{\}]) \wedge$ $\sigma_1'''(s) = []$ | eval-ops(4) |
| 7 | from $\sigma_1''' : \Sigma, s' : E;$ $(rtn, \sigma_1'', \pi' \curvearrowright [s' \cup es]) \xrightarrow{rtn} (\sigma_1''', \pi' \curvearrowright [s' \cup es, \{\}]) \wedge \sigma_1'''(s) = []$ | |
| 7.1 | $\sigma_1'''(a).status = \mathrm{INDS}$ | is-prem-dySt |
| 7.2 | $s = \sigma_1'''(a).cs$ | enter s (h5), not left yet |
| 7.3 | $\sigma_1'''(cs) = []$ | is-prem-dySt |
| | let $\pi'' = \pi' \curvearrowright [s' \cup es, \{\}]$ | |
| | infer $(rtn, \sigma_1''', \pi'') \xrightarrow{s} (\sigma_1''' \dagger \{a \mapsto mk_act(_, _, \mathrm{WAIT_TR})\}, \{\})$ | conc-dySt(7.1,7.3) |
| 8 | $(rtn, \sigma_1''', \pi'') \xrightarrow{s} (\sigma_1''' \dagger \{a \mapsto mk_act(_, _, \mathrm{WAIT_TR})\}, \{\})$ | $\exists$-E(6,7) |
| 9 | $(rtn, \sigma_1''' \dagger \{a \mapsto mk_act(_, _, \mathrm{WAIT_TR})\}, \pi'' \curvearrowright [s'' \cup \{\}]) \xrightarrow{rtn} (\sigma_2, \pi'' \curvearrowright [s'' \cup \{\}, \{\}])$ | RTN-Tick |
| | let $\pi''' = \pi'' \curvearrowright [s'' \cup \{\}, \{\}]$ | |
| 10 | $\exists tr \in RTN(a).sm.ts \cdot (tr, \sigma_2, \pi''') \xrightarrow{tr} \mathbf{true}$ | must at least be one tr valid by wf-rtn |
| 11 | $\mathsf{len}\, \pi''' < t_1 + wcet$ | prem-tr-stSt |
| 12 | $(rtn, \sigma_2, \pi''') \xrightarrow{s} (\sigma_2', \{\downharpoonleft s, \upharpoonleft tr.trg\})$ | conc-tr-stSt |
| 13 | $(rtn, \sigma_2', \pi''' \curvearrowright [s''' \cup \{\downharpoonleft s, \upharpoonleft tr.trg\}]) \xrightarrow{rtn} (\sigma_2'', \pi''' \curvearrowright [s''' \cup \{\downharpoonleft s, \upharpoonleft tr.trg\}, \{\}])$ | RTN-Tick |
| | let $\pi_2 = \pi''' \curvearrowright [s'' \cup \{\downharpoonleft s, \upharpoonleft tr.trg\}, \{\}]$ | |
| | let $t_2 = \mathsf{len}\, \pi_2$ | |
| 14 | $\downharpoonleft s \in_i \pi_2(t_2)$ | |
| | infer $\exists t_2 \in \mathsf{inds}\, \pi \cdot \downharpoonleft s \in_i \pi(t_2)$ | $\exists$-I($t_2$,14) |

■

## 6.5 Review

The soundness proofs we have presented in this chapter are based upon existing techniques for operational semantics. We have presented a model space which our SOS semantics define and the axiomatic semantics should be sound with respect to. The level of *rigour* of the soundness proofs is sufficient to provide a convincing argument that the axiomatic semantics are sound. However, it is obvious to state that a formal proof would extend the evidence as would a mechanised proof in an automated proof assistant, such as Isabelle/HOL.

The first attempt to formalise this style of approach was described in [CM92], which described a suite of HOL tools for the definition of a set of functions which took as an argument a list of formation rules and automatically proves the defining properties of the relation inductively defined by them. More precisely, this derived HOL inference rule builds a term that denotes the smallest relation closed under those rules. Nipkow [Nip98] formalised the opening 100-pages of [Win93] from which our approach in Section 6.2.3 is based upon. This formalisation was undertaken using the Isabelle/HOL proof assistant.

The structure and formal model underpinning our proofs is influenced by the work of [CM92, Nip98, Win93] such that we inductively define over the derivations our operational semantics.

Once the reader is convinced by the validity of our induction arguments of the RTN semantics, then the proofs for each axiom follow quite easily. By providing such induction principles we have allowed for concentration at the primary task: to give a concise and constructive operational semantics for RTNs. This allows for clear presentation of the concurrent and interleaving model, and the fine detail of each *RTN Action* separately, to avoid undue confusion or mistake.

# Part III

# Towards Tolerance

# Chapter 7

# Showing enhancements via graph grammars

## Contents


**7.1 Passive State Replication Template** . . . . . . . . . . 144
7.1.1 Description . . . . . . . . . . 144
7.1.2 Transformation . . . . . . . . . . 146
7.1.3 Design . . . . . . . . . . 151
7.1.4 Rigorous Proof . . . . . . . . . . 152
**7.2 Triple Modular Redundant Template** . . . . . . . . . . 156
7.2.1 Transformation . . . . . . . . . . 156
7.2.2 Design . . . . . . . . . . 157
7.2.3 Rigorous Proof . . . . . . . . . . 159
7.2.4 Permutations . . . . . . . . . . 160
**7.3 Watchdog Timer Template** . . . . . . . . . . 162
**7.4 Template Variations** . . . . . . . . . . 164
7.4.1 Temporal Redundant IDA . . . . . . . . . . 164
7.4.2 Fail-Signal (Activity) . . . . . . . . . . 166
7.4.3 Fail Stop (Activity) . . . . . . . . . . 167


In this Chapter, we detail several design templates of classical fault tolerant strategies appropriate for RTNs and RTN-SL designs. Each template design is given as a RTN-SL design fragment and a graph grammar representation which provides for a mechanism to instantiate each template into a design. For each template we give an informal description and then specify the fault assumptions, or *fault hypothesis*, made for each template. We attempt to follow [Bor98] as closely as possible which has found acceptance in the UK Defence industry for designs of RTNs to provide a standard template structure which is familiar to designers and engineers. Additionally, we give RTL theorems and proofs of the fault tolerant properties of each template under the given fault hypothesis. It is hoped these can then be used in the verification

| Events | Faults | Dynamic States | Static States | Activity |
|---|---|---|---|---|
| $\upharpoonleft s$ | early | | | |
| | late | | | |
| | value | | | |
| | omit | | | |
| | commit | | | |
| $\downharpoonleft s$ | early | *EarlyDyExit* | *EarlyStTimeExit* & *EarlyStEvExit* | |
| | late | *LateDyExit* | *LateStTimeExit* & *LateStEvExit* | |
| | value | | | |
| | omit | | | |
| | commit | | | |
| **rds** | early | *EarlyRead* | | |
| | late | *LateRead* | | *LateActRead* |
| | value | *ReadValue* | | |
| | omit | *OmitRead* | | |
| | commit | | | |
| **wds** | early | *EarlyWrite* | | |
| | late | *LateWrite* | | |
| | value | *WriteValue* | | |
| | omit | *OmitWrite* | | |
| | commit | | | |
| **progress** | Crash | *Crash* | | |

Table 7.1: Faults previously considered in Chapter 2

of designs, such that these theorems state the given functionality of the component the templates replace, when considering the fault-hypothesis.

We repeat Table 4.2 from Chapter 2 which illustrated the fault classes we consider for RTNs. The repeated table in Table 7.1 illustrates the coverage of the proposed templates for the faults considered.

Some templates seem very complicated for little reward. This is partly due to the design notation and the RTN architecture, which –for example– makes explicit each communication path. For some templates, the graph grammar presentation is straightforward, however the specification and semantics are more intricate, alternatively, those graph grammar presentations which seem overly complicated do have a more straightforward and simplistic semantic description.

In Chapter 3, we proposed to use graph grammars as our transformational methodology which *replaces* a component in a design with another –template of– component(s) that achieves some degree of fault tolerance. Given this template has been the running example throughout the earlier chapters –especially Chapter 3– we conclude that example in Sections 7.1.2, 7.1.3 & 7.1.4 which details the fault-hypothesis, RTL formulas and proofs of, the fault tolerant properties this template provides.

## 7.1 Passive State Replication Template

### 7.1.1 Description

The first template we consider is termed *passive state replication* (PSR). Though this template seems straightforward (given that activities in RTNs are sequential) it represents the extent of **replication** possible at the activity level. However, this template provides for several fault tolerant masking solutions to a broad array of faults. This replication strategy offers timing and omission tolerance internally to an activity. This is achieved by using the extended RTN-SL language to specify *fault transitions* in a state-machine specification. For *passive state replication* and a fault hypothesis of timing and omission faults, we specify transitions for *late*, *early* and *omission* events either for ports (read/write/stim) or state-transitions (entry/exit) actions. This template, in its current -intentional- generic format can tolerate:

- Late reads at port, *pA;*
- Omission faults at port, *pA*

We clarify the range of faults the PSR template tolerates with respect to the context a transformation is proposed for. For the example shown in Figure 7.1, the dynamic state we propose to transform reads an input and writes a result. It is a requirement that the transformation -at is neighbourhood- maintains this context and is therefore *interface preserving.*

### 7.1.2 Transformation

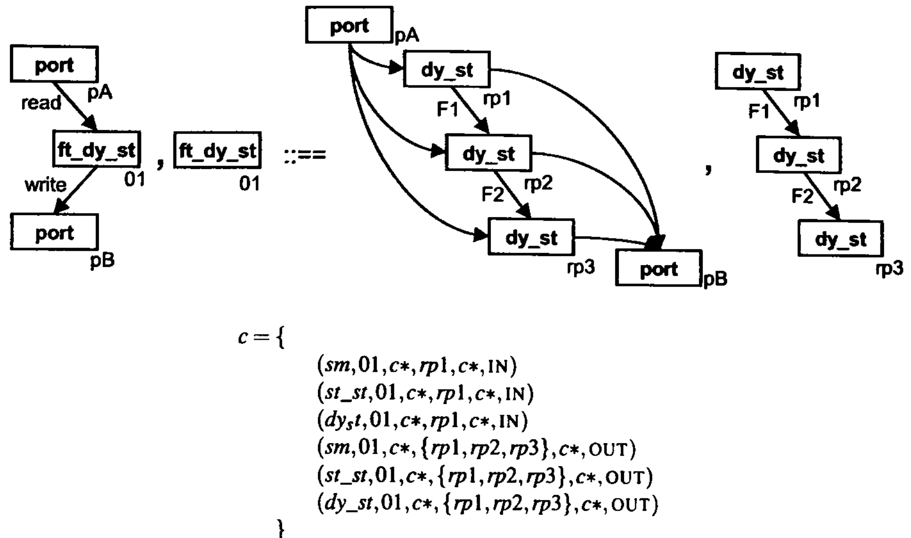
#### Graph Grammar Representation

Four permutations can exist for the *passive state replication* template. We elect to describe the case that the existing component reads and writes an input/output respectively. The other cases are that a component only reads an input; only writes an output; or does neither. The case we have chosen covers three of the four cases so is ideal to describe concisely the detail.

Given the original component reads and writes the template must similarly do so. Again, permutations of *the permutations* could be laboured here, given the various choices (Pool, Signal, Channel or Constant) of each data port. However, we elect to describe the case that the input port is a channel and the output be a pool to highlight the issues the protocols raise.

The characteristics of an incoming channel and an outward pool are that the reader may be held up whereas the writer has no such restriction. In addition to the protocol blocking a reader, a *read action* would also consume the data which is an important consideration for the PSR template. These characteristics motivate two design choices for the template:

- Replica's *rp1* & *rp2* must *speculatively* read, replica *rp3* –as originally did– reads *normally*[1];
- Only one replica will succeed in writing to the port as only one replica can succeed




$c = \{$

$(sm, 01, c*, rp1, c*, \text{IN})$
$(st_st, 01, c*, rp1, c*, \text{IN})$
$(dy_st, 01, c*, rp1, c*, \text{IN})$
$(sm, 01, c*, \{rp1, rp2, rp3\}, c*, \text{OUT})$
$(st_st, 01, c*, \{rp1, rp2, rp3\}, c*, \text{OUT})$
$(dy_st, 01, c*, \{rp1, rp2, rp3\}, c*, \text{OUT})$

$\}$

Figure 7.1: Template #1 - Passive state replication

The embedding relation ($c$) in Figure 7.1 specifies how the neighbourhood of the parent in the host graph is re-linked to the daughter in the rest graph. Here, we have used some notational freehand to say that, on the incoming arcs, each node connected to 01 in the parent graph should be connected to $rp1$ with the same label. The context graph $B$ (the second graph in Figure 7.1) specifies how the daughter graph is connected to the context graph $A$ of the mother (the first graph in Figure7.1 ).

### Fault Hypothesis

Given the specification for a *late write* fault related to the input at $pA$, the initial fault may have been either the input (at $pA$) or the computation within the dynamic state to generate the output to $pB$. However, as no mention is made in the fault hypothesis of any fault at $pA$, we must therefore examine that a dynamic state, given a timely input, fails to produce its output timely.

A fault that causes a component to produce the expected value for a given nonempty input sequence too late will be termed a *late timing* fault, which in RTL can be defined as:

$$LATE_FAULT : Event \times Occ \times Interval \rightarrow \mathbb{B}$$
$$LATE_FAULT(e, i, I) \triangleq$$
$$\exists t' : Time \cdot t' \geq I.u \wedge \Theta(e, i, t')$$

The basic RTL formula states that an event, $e$ occurs for some occurence, $i$ at some time, $t$. Th $LATE_FAULT$ formula states that an event ($e$) occurs for the of the interval ($I$). This auxiliary function (and the fault axiom stated below) are discussed further in Section 4.2 when the Real-Time Logic (RTL) is introduced. The aim of the auxiliary function is to state the characteristics of the specific fault (Similar functions are defined for Early, Omit and Commit faults) so that the fault axioms can clearly identify the observable interval an event should occur within.

[1]The default read action *normally* is a destructive read which may be held-up.

Specific to our example a late write at port $p4$, in RTL, is specified to identify the time the read event occured (for the same occurence) and that the write event occurs later than some deadline ($X$).

$$
\begin{aligned}
&LateWrite(p4, i, t) \triangleq \\
&\quad \exists t' : Time \cdot t' \leq t \wedge \Theta(R_{p1}, i, t') \wedge \\
&\qquad\qquad\qquad LATE_FAULT(W_{p4}, i, [t', t' + X])
\end{aligned}
$$

An alternative definition may have been to relate the write event to another event. Such causal relationships are supported by the RTN-SL semantics, where –for example– the link axioms state:

$$\forall t : Time \cdot (\exists i : Occ \cdot \Theta(W_{p4}, i, t)) \Leftrightarrow \exists j : occ \cdot \Theta(\downarrow rp_1, j, t)$$

This casual relationship is used later for our reasoning. We therefore consider *late exit* faults in our reasoning.

#### RTN-SL Design Fragment

The RTN-SL specification of the template is shown below. We highlight the specification details with regards the incoming port ($pA$) and the fault transitions between replica's.

The transformation necessitates we read *speculatively* from $pA$ at replica's $rp1$ and $rp2$ to guarantee the same value is available for the subsequent replicas should one fail. This necessitates the protocol at $pA$ be changed from a normal one to a speculative one on the readers side. This does not however break the interface requirement that the interface remain constant as we only propose changing the readers side. We are however required to show the blocking and destructive behaviour is retained by the transformation. Should replica's $rp1$ or $rp2$ succeed, then we must destructively read from $pA$ before the next occurrence of entering $rp1$. Given $pA$ is speculative, no wait state (static state) is required and we know we can read from $pA$ if $rp1$ has succeeded (and no omission fault at $pA$ has occurred), we therefore perform a *flush* type operation to $pA$ where we (destructively) read a value and simply perform no computation.

The essence of the PSR template are its *fault transitions* specifications. The template states that should a late write fault be observed at $rp1$ then the RTN should transition to $rp2$, similarly that $rp2$ should transition to $rp3$. However, the assumption at most two of the replica's will fail prohibits $rp3$ from failing, therefore only a successful transition exists from $rp3$. Given the fault must occur within the dynamic state, we specify the fault transition as an exit transition. However, should a fault not occur then condition $c2$ must hold and therefore a timely write is made to $pB$. We note that only one write can occur to $pB$ for each occurrence that we enter $rp1$.

**Thesis–temp1.rtnsl**

```
adt dtl is
  type A is token;
  type B is token;
end adt;

activity templ is
  with dtl;

  ports
     pA : (Channel, dtl.A, in, SPECULATIVE);
     pB : (Pool, dtl.B, out);
  end ports;

  auxiliary definitions
     constant l_rpl : Time;
     constant l_rp2 : Time;
     constant l_rp3 : Time;
     constant wcet_rpl : Time;
     constant wcet_rp2 : Time;
     constant wcet_rp3 : Time;
     constant u_rpl : Time;
     constant u_rp2 : Time;
     constant u_rp3 : Time;
  end auxiliary definitions;

  operations
     op foo (input : misc.A);
       pre true;
       post true;
     end op;

     op foo_rpl (input_rpl : misc.A) output_rpl : misc.B;
       pre true;
       post true;
     end op;

     op foo_rp2 (input_rp2 : misc.A) output_rp2 : misc.B;
       pre true;
       post true;
     end op;

     op foo_rp3 (input_rp3 : misc.A) output_rp3 : misc.B;
       pre true;
       post true;
     end op;
  end operations;

  states
     dynamic rpl
       op foo_rpl peeps from pA writes to pB;
       timing [l_rpl,wcet_rpl,u_rpl];
       Fl => transition goes to rp2;
       transition goes to flush on c2;
     end state;

     dynamic rp2
       op foo_rpl peeps from pA writes to pB;
       timing [l_rp2,wcet_rp2,u_rp2];
       Fl => transition goes to rp2;
       transition goes to flush on c2;
     end state;

     dynamic rp3
       op foo rpl reads from pA writes to pB;
       timing [l_rp3,wcet_rp3,u_rp3];
       transition goes to flush on c2;
     end state;

     dynamic flush
       op foo reads from pA;
       timing [l,wcet,u];
       transition goes to rpl on true;
```

**Thesis–temp1.rtnsl**

```
     end state;

     initial rpl;
   end states;

end activity;

rtn templ is
end rtn;

theory

  Theorem : THEOREM
     FORALL (input : dtl.a, output : dtl.b):
        pre_foo(input) AND post_foo(input,output) IMPLIES
                       (pre_foo_rpl(input) AND post_foo_rpl(input,output) AND
                       pre_foo_rp2(input) AND post_foo_rp2(input,output) AND
                       pre_foo_rp3(input) AND post_foo_rp3(input,output))

end;
```

### 7.1.3 Design

This section concludes the formal reasoning of the running example which began in Chapter 3. We have seen above the transformation proposed and its specification to tolerate the fault hypothesis identified. Below we give the modified semantics (instantiated from $\Omega_f$) from that given in [PAH00]. We have retained the same naming convention (except that our activities are named *A* and *B* instead of *A1* and *A2* respectively) to illustrate the changes.

The example introduced in Chapter 3 was the faulty behaviour of a single dynamic state that was hypothesised to potentially write late. We then illustrated how the *passive state replication* template can be applied to the host design to tolerate this behaviour. The template design essentially replaced a single dynamic state with an arrangement of three replica's. The template allows for –at most– two faults which define the state-machine transitions within the template, and therefore activity. The generated semantics differ from the original at the interface between the components, for example, the state-machine transition target states are now the replicas. Within the template, the transitions are dependant on the replicas' behaviour.

The modified theory for activity B is now:

**B2_D:** $\forall i : Occ, t : Time \cdot \Theta(\downarrow D, i, t) \;\Rightarrow$
$\qquad \Theta(\uparrow E'_rp1, i, t)$

**B3_E'_rp1:** $\forall i, k : Occ, t : Time \cdot \Theta(\downarrow E'_rp1, i, t) \;\Rightarrow$
$\qquad ((LATE_FAULT(\downarrow E'_rp1, i, [t\text{-}rp1_u, t\text{-}rp1_l]) \wedge$
$\qquad\qquad \exists j : Occ \cdot \Theta(\uparrow E'_rp2 j, t)) \vee$
$\qquad (\neg FAULT \wedge \Phi(true, k, t) \wedge \Theta(\uparrow D, i+1, t)))$

**B3_E'_rp2:** $\forall i, k : Occ, t : Time \cdot \Theta(\downarrow E'_rp2, i, t) \;\Rightarrow$
$\qquad ((LATE_FAULT(\downarrow E'_rp2, i, [t\text{-}rp2_u, t\text{-}rp2_l]) \wedge$
$\qquad\qquad \exists j : Occ \cdot \Theta(\uparrow E'_rp3 j, t)) \vee$
$\qquad (\neg FAULT \wedge \Phi(true, k, t) \wedge \exists j : Occ \cdot \Theta(\uparrow D + 1, j, t)))$

**B3_E'_rp3:** $\forall i, k : Occ, t : Time \cdot \Theta(\downarrow E'_rp3, i, t) \;\Rightarrow$
$\qquad \neg FAULT \wedge \Phi(true, k, t) \wedge \exists j : Occ \cdot \Theta(\uparrow D + 1, j, t)$

**B4_D:** $\forall i, j : Occ, t : Time \cdot \Theta(\uparrow D, i, t) \;\Rightarrow$
$\qquad (i = 0 \vee$
$\qquad\qquad ((i > 1 \wedge (\Theta(\downarrow E'_rp1, j, t) \vee$
$\qquad\qquad\qquad \Theta(\downarrow E'_rp2, j, t) \vee$
$\qquad\qquad\qquad \Theta(\downarrow E'_rp3, j, t))$
$\qquad\qquad ))$
$\qquad )$

**B5_D:** $\forall i : Occ, t : Time \cdot \Theta(\uparrow E'_rp1, i, t) \;\Rightarrow$
$\qquad \Theta(\downarrow D, i, t)$

**B6_E'_rp1:** $\forall i : Occ, t : Time \cdot \Theta(\uparrow E'_rp1, i, t) \;\Rightarrow$
$\qquad ((\exists t' : Time \cdot t + l2 \leq t' \leq t + u2 \wedge \Theta(\downarrow E'_rp1, i, t')) \vee$
$\qquad (LATE_FAULT(\downarrow E'_rp1, i, [t + rp1_l, t + rp1_u] \wedge$
$\qquad\qquad \exists t'' : Time \cdot t'' > t + rp1_u \wedge \Theta(\downarrow E'_rp1, i, t'')))$

**B6_E'_rp2:** $\forall i : Occ, t : Time \cdot \Theta(\uparrow E'_rp2, i, t) \Rightarrow$
$((\exists t' : Time \cdot t + l2 \leq t' \leq t + u2 \wedge \Theta(\downarrow E'_rp2, i, t')) \vee$
$LATE_FAULT(\downarrow E'_rp2, i, [t + rp2_l, t + rp2_u] \wedge$
$\exists t'' : Time \cdot t'' > t + rp2_u \wedge \Theta(\downarrow E'_rp2, i, t'')))$

**B6_E'_rp3:** $\forall i : Occ, t : Time \cdot \Theta(\uparrow E'_rp3, i, t) \Rightarrow$
$(\exists t' : Time \cdot t + l2 \leq t' \leq t + u2 \wedge \Theta(\downarrow E'_rp3, i, t')$

**B9_E'_rp1:** $\forall i : Occ, t : Time \cdot \Theta(\downarrow E'_rp1, i, t) \Rightarrow$
$((\exists t' : Time \cdot t + rp1_l \leq t' \leq t + rp1_u \wedge \Theta(\uparrow E'_rp1, i, t')) \vee$
$LATE_FAULT(\downarrow E'_rp1, i, [t\text{-}rp1_u, t\text{-}rp1_l]))$

**B9_E'_rp2:** $\forall i : Occ, t : Time \cdot \Theta(\downarrow E'_rp2, i, t) \Rightarrow$
$((\exists t' : Time \cdot t + rp2_l \leq t' \leq t + rp2_u \wedge \Theta(\uparrow E'_rp2, i, t')) \vee$
$LATE_FAULT(\downarrow E'_rp2, i, [t\text{-}rp2_u, t\text{-}rp2_l]))$

**B9_E'_rp3:** $\forall i : Occ, t : Time \cdot \Theta(\downarrow E'_rp3, i, t) \Rightarrow$
$(\exists t' : Time \cdot t + rp3_l \leq t' \leq t + rp3_u \wedge \Theta(\uparrow E'_rp3, i, t')$

### 7.1.4 Rigorous Proof

The original presentation of our example in [PAH00] gave two proofs to show the end-to-end specifications are satisfied by the design. We now investigate proving the same theorems to show the transformed –more fault tolerant– design satisfies the specification under the assumed fault hypothesis and assumption that at most only two components can fail. We proceed to suggest a modification to the proofs that retain the original structure, which is desired of our transformational design methodology.

The two theorems we wish to prove are Theorem 3.1 and Theorem 3.2. Theorem 3.1 crucially depended on a lemma that activity *B steps*: given an input to $p3$ an output is produced to $p4$ within the specified time bounds. This is specified in Lemma 7.1. The original proof of this lemma showed that entering state $E$ on the $R_{p3}$ event meant leaving the state timely. We are therefore required to show an equivalent lemma holds for the template specification which is specified in Lemma 7.2.

**Lemma 7.1 (BSteps)** $\forall i : Occ, t_1 : Time \cdot \Theta(R_{p3}, i, t_1) \Rightarrow \exists t_2 \cdot \Theta(\downarrow E, i, t_2) \wedge t_1 + u_2 \geq t_2 \geq t_1 + l_2$

**Lemma 7.2 (Temp2Steps)** $\forall i : Occ, t_1 : Time \cdot \Theta(\uparrow E', i, t_1) \Rightarrow \exists t_2 : Time \cdot \Theta(\uparrow D, i + 1, t_2) \wedge t_1 + X \geq t_2$

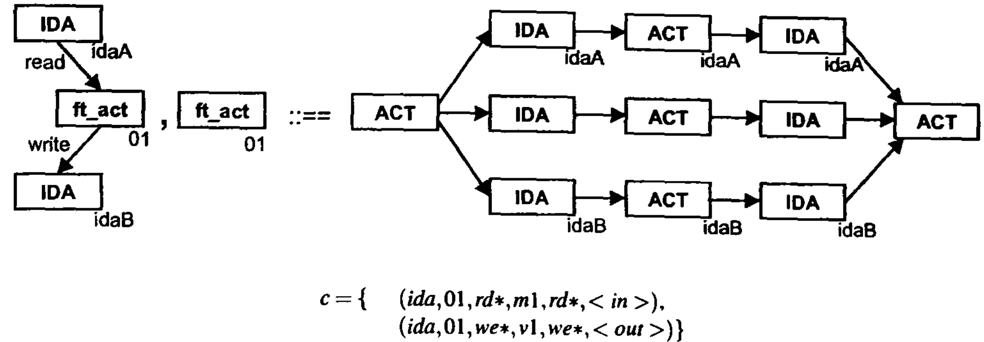
Note, Lemma 7.2 asserts entering state $D$ ($\uparrow D$) for the next occurrence rather than leaving state $E(\downarrow E)$, but it is clear from the semantics these are equivalent.

PROOF Informally, Lemma 7.2 can be seen to hold by realising that an output must have occurred if Activity $B$ enters state $D$ for the next occurrence, which in turn can only occur if $E'$ is entered because of an attempt to read data from the channel. As state $D$ is entered for the next occurrence then at most only two faults could have occurred; if only one fault had occurred, then the exit from *rp1* would have transitioned to *rp2* which must have succeeded and transitioned, in turn, to state $D$; if two faults had occurred then *rp2* would have transitioned similarly to *rp3* then in turn to state $D$; if no faults occurred then *rp1* would transition similarly to state $D$ again.

| | | |
|---|---|---|
| | from Activity B Theory | |
| 1 | from $i : Occ, t_2 : Time, \Theta(\Downarrow D, i, t_2)$ | |
| 1.1 | $\Theta(\Uparrow E'_rp1, i, t_2)$ | B2_D(1) |
| 2 | from $LATE_FAULT(\Downarrow E'_rp1, i, [t_2 + E'_rp1_l, t_2 + E'_rp1_u])$ | FH |
| 2.2 | $\Theta(\Downarrow E'_rp1, i, t_3) \wedge t_3 > t_2 + E'_rp1_u$ | B6_E'_rp1(1.1) |
| 3 | $\Theta(\Uparrow E'_rp2, j, t_3)$ | B3_E'_rp1(2.2) |
| 4 | from $LATE_FAULT(\Downarrow E'_rp2, j, [t_3 + E'_rp2_l, t_3 + E'_rp2_l])$ | FH |
| 4.1 | $\Theta(\Downarrow E'_rp2, j, t_4) \wedge t_4 > t_3 + E'_rp2_u$ | B6_E'_rp2(3,4) |
| 4.2 | $\Theta(\Uparrow E'_rp3, k, t_4)$ | B3_E'_rp2(4.1,4) |
| 4.3 | $\Theta(\Downarrow E'_rp3, k, t_5) \wedge t_4 + E'_rp3_u \geq t_5 \geq t_4 + E'_rp3_l$ | B6_E'_rp3(4.2,4) |
| 4.4 | $\Theta(\Uparrow D, i+1, t_5)$ | B3_E'_rp3(4.3,4) |
| 5 | infer $LATE_FAULT(\Downarrow E'_rp2, j, [t_3 + E'_rp2_l, t_3 + E'_rp2_u]) \Rightarrow$ $\exists t_5 : Time \cdot \Theta(\Uparrow D, i+1, t_5) \wedge t_4 + E'_rp3_u \geq t_5 \geq t_4 + E'_rp3_l$ | ∃-intro(4.4) |
| 6 | from $\neg LATE_FAULT(\Downarrow E'_rp2, j, [t_3 + E'_rp2_l, t_3 + E'_rp2_u])$ | |
| 6.1 | $\Theta(\Downarrow E'_rp2, j, t_6) \wedge t_3 + E'_rp2_u \geq t_6 \geq t_3 + E'_rp2_l$ | B6_E'_rp2(3,6) |
| 6.2 | $\Theta(\Uparrow D, i+1, t_6)$ | B3_E'_rp2(6.1,6) |
| 7 | infer $\neg LATE_FAULT(\Downarrow E'_rp2, j, [t_3 + E'_rp2_l, t_3 + E'_rp2_u]) \Rightarrow$ $\exists t_6 : Time \cdot \Theta(\Uparrow D, i+1, t_6) \wedge t_3 + E'_rp2_u \geq t_6 \geq t_3 + E'_rp2_l$ | ∃-intro(6.2) |
| 8 | $\Theta(\Uparrow D, i+1, t_7) \wedge t_7 \geq t_6 \geq t_5$ | ∨-E(5,7) |
| 10 | infer $LATE_FAULT(\Downarrow E'_rp1, i, [t_2 + E'_l, t_2 + E'_u]) \Rightarrow$ $\exists t_7 : Time \cdot \Theta(\Uparrow D, i+1, t_6) \wedge t_7 \geq t_6 \geq t_5$ | ∃-intro(8) |
| 11 | from $\neg LATE_FAULT(\Downarrow E'_rp1, i, [t_2 + E'_rp1_l, t_2 + E'_rp1_u])$ | |
| 11.1 | $\Theta(\Downarrow E'_rp1, i, t_8) \wedge t_2 + E'_rp1_u \geq t_8 \geq t_2 + E'_rp1_l$ | B6_E'_rp1(1.1,11) |
| 11.2 | $\Theta(\Uparrow D, i+1, t_8)$ | B3_E'_rp1(1.2,11) |
| 12 | infer $\neg LATE_FAULT(\Downarrow E'_rp1, i, [t_2 + E'_rp1_l, t_2 + E'_rp1_u]) \Rightarrow$ $\exists t_8 : Time \cdot \Theta(\Uparrow D, i+1, t_8) \wedge t_2 + E'_rp1_u \geq t_8 \geq t_2 + E'_rp1_l$ | ∃-intro(11.2) |
| 13 | $\Theta(\Uparrow D, i+1, t_9) \wedge t_9 \geq t_7 \geq t_8$ | ∨-E(10,12) |
| 14 | infer $\Theta(\Downarrow E', i, t_2) \Rightarrow \exists t_9 : Time \cdot \Theta(\Uparrow D, i+1, t_9) \wedge t_2 + MAX \geq t_9$ | ∃-into(14) |
| | infer $\forall i : Occ, t_2 : Time \cdot \Theta(\Downarrow E', i, t_2) \Rightarrow$ $\exists t_9 : Time \cdot \Theta(\Uparrow D, i+1, t_9) \wedge t_2 + MAX \geq t_9$ | ∀-intro(14) |

■

The essence of the proof was in establishing the conjecture that from leaving state $D$ for the $ith$ occurrence, there exists a time later that we enter state $D$ for the $i+1th$ occurrence. We proved this conjecture in the presence of faults, which therefore necessitated we consider the disjunction that a fault does, or does not, occur at each replica (though the assumption is the third replica cannot fail).




$$c = \{ \quad (ida, 01, rd*, m1, rd*, < in >), \\ (ida, 01, we*, v1, we*, < out >)\}$$

Figure 7.2: Template #2 - Triple Modular Redundancy

## 7.2 Triple Modular Redundant Template

The second template we consider for RTNs is termed *Triple Modular Redundancy* (TMR). This strategy employs three concurrent replicas to each calculate an output given identical inputs. Each replica is passed the input to the parent node from a *multicaster* which it then processes. Each replica then writes its result to a *voter* which tests for consensus of the outputs. At least two outputs must agree. This template therefore provides for value, timing and omission (or crash) tolerance. Permutation of this template are discussed in Section 7.2.4.

### 7.2.1 Transformation

Figure 7.2 illustrates the graph grammar presentation which shows, in the context of one input path and one output path, the *ft_act* (i.e. the activity to replicate) is replaced by an arrangement of the three replicas and the mentioned multicaster and voter. It is the multicaster and voter that read, and write to, the original input and output paths, respectively. This presumes the interface requirement, that no affect resonates to the surrounding network. Interestingly, the nature of the input and output protocols effects the design of the multicaster and voter components[2].

Although a context (graph) for the parent graph is specified for this template, no context (graph) is required for the daughter graph. Simply, the embedding relation ($c$) states that the IDA which the *ft_act* node reads from is read by the multicaster ($m1$) node, similarly the IDA the *ft_act* write to is written to by the voter ($v1$) node.

[2]It is feasible that graph grammar transformations can allow for the permutations of protocols by describing transformations for multicaster and voter components in the various contexts of input and output paths. These transformations however are not described here.

**Thesis–temp2.rtnsl**

```
adt dtl is
  type T1 is token;
end adt;

activity ml is
  with dtl;

 ports
    pA : (Channel, dtl.T1, in);
    pB' : (Channel, dtl.T1, out, SPECULATIVE);
    pB'' : (Channel, dtl.T1, out, SPECULATIVE);
    pB''' : (Channel, dtl.T1, out, SPECULATIVE);
 end ports;

  local state
        msg : dtl.T1;
  end local state;

  operations
        op multicast_data () processed : dtl.T1;
         ext read msg;
          pre true;
          post processed = msg~;
        end op;

        op read_data (raw : dtl.T1);
         ext write msg;
          pre true;
          post msg = raw;
        end op;

  end operations;

  states
        static stateA
          transition goes to stateB on read pA;
        end state;

        dynamic stateB
          op      read_data reads from pA;
          timing  [0,0,0];
          transition goes to stateC' on true;
        end state;

        dynamic stateC'
          op      multicast_data writes to pB';
          timing  [0,0,0];
          write failure => transition goes to stateC'' on true;
          write success => transition goes to stateC'' on true;
        end state;

        dynamic stateC''
          op      multicast_data writes to pB'';
          timing  [0,0,0];
          write failure => transition goes to stateC''' on true;
          write success => transition goes to stateC''' on true;
        end state;

        dynamic stateC'''
          op      multicast_data writes to pB''';
          timing  [0,0,0];
          write failure => transition goes to stateA on true;
          write success => transition goes to stateA on true;
        end state;

    initial stateA;
 end states;

end activity;

activity vl is
  with dtl;
```

**Thesis–temp2.rtnsl**

```
 ports
    pl : (Pool, dtl.T1, in);
    p2 : (Pool, dtl.T1, in);
    p3 : (Pool, dtl.T1, in);
    p4 : (Pool, dtl.T1, out);
 end ports;

  local state
        msg_pl : dtl.T1;
        msg_p2 : dtl.T1;
        msg_p3 : dtl.T1;
        consensus : bool;
  end local state;

  operations
        op vote () processed : dtl.T1;
         ext read msg_pl, msg_p2;
          pre true;
          post (msg_pl=msg_p2 implies processed = msg_pl~ and consensus=true) or
               (msg_pl=msg_p3 implies processed = msg_pl~ and consensus=true) or
               (msg_p2=msg_p3 implies processed = msg_p2~ and consensus=true);
        end op;

        op read_data (raw : dtl.T1);
         ext write msg_pl;
          pre true;
          post msg_pl = raw;
        end op;

  end operations;

  states
        dynamic stateA
          op read_data reads from pl;
          op read_data reads from p2;
          op read_data reads from p3;
          timing  [0,0,0];
          transition goes to stateV on true;
        end state;

        dynamic stateV
          op vote writes to p4;
          timing [0,0,0];
          transition goes to stateA on consensus = true;
          transition goes to stateT on consensus = false;
        end state;

        static stateT
        end state;

        initial stateA;
  end states;

end activity;

activity rpl is
  with dtl;

 ports
    pl : (Channel, dtl.T1, in);
    p2 : (Pool, dtl.T1, out);
 end ports;

 operations
        op foo (input : dtl.T1) output : dtl.T1;
          pre true;
          post true;
        end op;
 end operations;

 states
```

**Thesis-temp2.rtnsl**

```
      static stateA
        transition goes to stateB on read p1;
      end state;

      dynamic stateB
        op foo reads from p1 writes to p2;
        timing [0,0,0];
        transition goes to stateA on true;
      end state;

      initial stateA;
 end states;

end activity;


activity rp2 is
  with dtl;

 ports
    p1 : (Channel, dtl.T1, in);
    p2 : (Pool, dtl.T1, out);
 end ports;

 operations
       op foo (input : dtl.T1) output : dtl.T1;
         pre true;
         post true;
       end op;
 end operations;

 states
      static stateA
        transition goes to stateB on read p1;
      end state;

      dynamic stateB
        op foo reads from p1 writes to p2;
        timing [0,0,0];
        transition goes to stateA on true;
      end state;

      initial stateA;
 end states;

end activity;

activity rp3 is
  with dtl;

 ports
    p1 : (Channel, dtl.T1, in);
    p2 : (Pool, dtl.T1, out);
 end ports;

 operations
       op foo (input : dtl.T1) output : dtl.T1;
         pre true;
         post true;
       end op;
 end operations;

 states
      static stateA
        transition goes to stateB on read p1;
      end state;

      dynamic stateB
        op foo reads from p1 writes to p2;
        timing [0,0,0];
        transition goes to stateA on true;
      end state;
```

**Thesis-temp2.rtnsl**

```
      initial stateA;
 end states;

end activity;

ida m1_ida is
  Kind SPECULATIVE Channel;
  Datatype dtl.T1;
end ida;

ida m2_ida is
  Kind  SPECULATIVE Channel;
  Datatype dtl.T1;
end ida;

ida m3_ida is
  Kind SPECULATIVE Channel;
  Datatype dtl.T1;
end ida;

ida v1_ida is
  Kind Pool;
  Datatype dtl.T1;
end ida;

ida v2_ida is
  Kind Pool;
  Datatype dtl.T1;
end ida;

ida v3_ida is
  Kind Pool;
  Datatype dtl.T1;
end ida;

rtn templ is
  m1.pB' writes to m1_ida;
  m1.pB'' writes to m2_ida;
  m1.pB''' writes to m3_ida;
  rp1.p1 reads from m1_ida;
  rp1.p2 writes to v1_ida;
  rp2.p1 reads from m2_ida;
  rp2.p2 writes to v2_ida;
  rp3.p1 reads from m3_ida;
  rp3.p2 writes to v3_ida;
  v1.p1 reads from v1_ida;
  v1.p2 reads from v2_ida;
  v1.p3 reads from v3_ida;
end rtn;

theory

end;
```

### Fault Hypothesis

Again, from the RTL definition of faults for RTNs in Section 4.2, we can define a value and crash fault (specified in RTL) as:

$$
\begin{aligned}
&WriteValue(p,i,t) \triangleq \\
&\quad \exists t' : Time \cdot t' < t \wedge \Theta(\uparrow s, i, t') \wedge \Theta(\downarrow s, i, t) \wedge \\
&\qquad\qquad pre_(s.in_p(t'), \overleftarrow{v}(t')) \wedge \\
&\qquad\qquad \neg post_(s.in_p(t'), \overleftarrow{v}(t'), v(t), s.out_p(t))
\end{aligned}
$$

The second fault hypothesis we consider for the template is a *crash* fault, but as stated in Section 4.2.1.2 we map activity faults to their state-machine counterparts. Therefore, the abstract definition of a *crash activity* fault is:

$$
\begin{aligned}
&Crash(s,i,t) \triangleq \\
&\quad \exists t' : Time \cdot \Theta(\uparrow s, i, t') \wedge \\
&\qquad\qquad OMIT(\downarrow s, i, [t' + s.bcet, t]) \wedge \\
&\qquad\qquad OMIT(\uparrow s, i+1, [t' + s.bcet, t])
\end{aligned}
$$

which in turn is specified as:

$$
\begin{aligned}
&[is_Activity(act)] \\
&CrashAct(act,i,t) \triangleq \\
&\quad \exists s \in act.SM.states \cdot Crash(s,i,t)
\end{aligned}
$$

We consider both these definitions as our fault hypotheses when reasoning about this template.

### Axiomatic Semantics

The axiomatic semantics that define this template are generated by the NetSpec tool for the specification given above. Examples of the semantics generated for a TMR template can be found in Chapter 8 when this template is used in the case study.

## 7.2.3 Rigorous Proof

The theorems we wish to show of this template, which will be used when reasoning about the fault tolerant properties of this template are that which state the liveliness of the template and the behaviour of the voter component.

**Lemma 7.3 (TMR-Liveliness)** $\forall i : Occ, t_1 : Time \cdot \Theta(R_{pA}, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 \leq t_1 + u \wedge \Theta(W_{p4}, i, t_2)$

**Lemma 7.4 (TMR-Value)** $\forall i : Occ, t : Time \cdot \Theta(rd_pA, i, t) \Rightarrow \exists t' : Time \cdot t' > t \wedge \Theta(we_p4, i, t')$

Informally, Lemma 7.4 states that an output should be produced, for each value read by the multicaster component, by the voter. For a value to be written as output from the voter, it must agree with at least one other value produced by a replica as specified in the specification. We therefore define this *value* domain fault in terms of observable events.

PROOF (PROOF OF LEMMA 7.4)

from $Assumptions, ActivityTheorems, LinkandChannelAxioms$

1 from $i : Occ, t_1 : Time, \Theta(rd_pA, i, t_1)$

1.1 $\Theta(we_m1, pB', t_6); \Theta(we_m1, pB'', t_7); \Theta(we_m1, pB''', t_8)$

...

1.2 $\Theta(we_rp1.p2, i, t_3)$

1.3 $\Theta(we_rp2.p2, i, t_4)$

1.4 $\Theta(we_rp3.p2, i, t_5)$

1.5 $\Theta(rd_v1.p1, i, t_3) \wedge \Theta(rd_v1.p2, i, t_4) \wedge \Theta(rd_v1.p3, i, t_5)$

1.5 $\Theta(we_p4, i, t_2) \wedge t_2 > t_1$

infer $\exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p4, i, t_2)$ — $\exists$-I(1.5)

2 $\Theta(rd_pA, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p4, i, t_2)$ — $\Rightarrow$-I(1)

infer $\forall i : Occ, t_1 : Time \cdot \Theta(rd_pA, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p4, i, t_2)$ — $\forall$-I(2)

■

### 7.2.4 Permutations

In this section we have illustrated only one permutation of the TMR strategy. Several characteristics define a permutation, notably the protocol characteristics (blocking or destructive read and writes) at the input and output ports.

The permutations are determined by the communication paths (IDAs) and their associated protocols of the incoming and outward IDAs as specified in the context graph. For example, if an activity reads from a *channel* protocol (as in the template above) then the three active replicas must all receive the same value (at the same time) to concurrently execute the replicated design to –hopefully– produce agreeable results. However, a *channel* protocol states a reader is

- blocked should no data be available to the read; and
- any read operation is destructive.

Therefore, each replica could not read from the same IDA (should it be possible[3]) and be guaranteed the same result (we do not wish that the writer to the IDA must write its value three times as this then extends the templates neighbourhood – a requirement we state should not occur of our transformational method). Therefore, we must use additional components, such as *multicaster* (and later *voters*) to read from IDAs and communicate to each replica the value read (similarly, a voter reads each result and writes the agreed result to the original IDA). Furthermore, the design of a multicaster and voter is dictated by the protocol.

The effect to the design of multicasters and voters is complicated, as mentioned, by the protocols. Therefore, when a multicaster reads from a channel, it must first *record* this value to its *local state* then write the value to each replica. The protocol we associate with this intermediate IDA –between multicaster and

[3]RTN-SL version 3.1 prohibits multiple readers and writes at IDAs

replica– must be the same as the multicaster reads from to preserve the stimulate behaviour to fire each replica in sync.

Fortunately, our approach in using graph grammars to describe and –more importantly– control the transformation, we can specify each permutation of the multicaster and voter components and the context they are applicable in. Furthermore, we can then specify the TMR template with non-terminal voter and multicaster nodes.

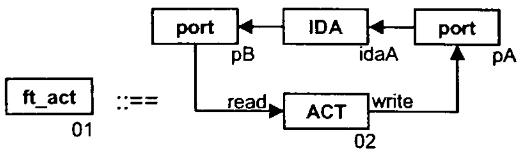



Figure 7.3: Template #3 - A Watchdog Timer

## 7.3 Watchdog Timer Template

The *watchdog timer* template is a simple arrangement of a single activity which has a loop back data connection to itself via an explicit IDA path. The template provides for (late) timing & omission fault tolerance only. The functionality (described in detail within its RTN-SL specification) of this template is to mask the timing & omission faults to a crash fault, but a critical application to write an output given some input, then the watchdog templates provides for a 'default' output within a deadline.

### Graph Grammar Representation

Figure 7.3 illustrates the graph grammar presentation of the watchdog template, including the embedding relation, *c*.

**Thesis-temp3.rtnsl**

```
adt dtl is
  type A is token;
  type B is token;
end adt;

activity temp3 is
  with dtl;

  ports
     pA : (Stimulus, dtl.A, out);
     pB : (Stimulus, dtl.A, in);
     p1 : (Channel, dtl.B, in);
     p2 : (Channel, dtl.B, out);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  operations
     op watchdog set : dtl.A;
       pre true;
       post true;
     end op;

     op foo (input : dtl.B);
       pre true;
       post true;
     end op;
  end operations;

  states
     static stateA
        transition goes to w1 on [l1,u1];
        transition goes to stateB on read p1;
     end state;

     dynamic w1
       op watchdog writes to pA;
       timing [l,wcet,u];
       transition goes to wait on true;
     end state;

     static wait
       transition goes to stateB on read p1;
       transition goes to stateC on stim pB;
     end state;

     dynamic stateB
        op foo reads from p1;
        timing [l,wcet,u];
        transition goes to stateA on false;
        transition goes to stateC on true;
     end state;

     dynamic stateC
        op critical writes to p2;
        timing [l,wcet,u];
        transition goes to term on true;
     end state;

     static term
     end state;

     initial stateA;
  end states;

end activity;

rtn temp3 is
```

**Thesis-temp3.rtnsl**

```
end rtn;

theory

  Theorem : THEOREM
     FORALL (i:Occ, t1:Time):
        (th(rd_p1,i,t1) AND
        (EXISTS (j:Occ): LateWrite(p2,j,t1) OR OmitWrite(p2,j,t1))) IMPLIES
               EXISTS (t2:Time): th(we_p2,i,t2) AND t2 <= t1 + X
end;
```

### Fault Hypothesis

The faults to which this template is tolerant, as defined in Section 4.2 are:

$$
\begin{aligned}
& LateWrite(p2, i, t) \triangleq \\
& \quad \exists t', t'' : Time \cdot t' \leq t \leq t'' \wedge \Theta(\uparrow s, i, t') \wedge \\
& \qquad \Theta(\downarrow s, i, t'') \wedge \\
& \qquad \mathsf{LATE}(wds(p2), i, [t' + s.bcet, t' + s.wcet])
\end{aligned}
$$

and

$$
\begin{aligned}
OmitWrite(p2, i, t) \exists t', t'' : Time \cdot t' \leq t \leq t'' \wedge \Theta(\downarrow s, i, t') \wedge \\
\Theta(\uparrow s, i+1, t'') \vee \Theta(wds(s.out_p), i+1, t') \wedge \\
\mathsf{OMIT}(we(p2), i, [t', t''])
\end{aligned}
$$

### RTL

The emergent property we assert for this template are:

$$
\begin{aligned}
& \forall i : Occ, t_1 : Time \cdot \Theta(rd_p1, i, t_1) \wedge (LateWrite(p2, i, t) \vee OmitWrite(p2, i, t)) \Rightarrow \\
& \qquad \exists(\cdot t_2 : Time) t_2 \leq t_1 + X \wedge \Theta(we_p2, i, t_2)
\end{aligned}
$$

That is, despite the normal RTN-SL deadlines, we assert a $we_p2$ will occur within some critical deadline.

## 7.4 Template Variations

We briefly introduced several more design templates which are applicable to RTN-SL designs. These templates are instantiated in the case study, so their description, specification and reasoning to not presented here in such detail as the templates above for space considerations.

### 7.4.1 Temporal Redundant IDA

The redundancy strategy termed *Temporal Redundant IDA* employs true concurrent data communication paths which are controlled by the *multicaster* and voter-like activity components. The difference between this strategy and TMR is that a value written is guaranteed to be read by its destined target rather than a correct output from a given input.

The tolerance characteristics of this template are: value (masking), omission & timing. Each value that is written to the intended *ft_ida* is guaranteed to be delivered to its recipient in a timely manner only if it agrees with the second value. Unlike for TMR, we can't vote on the correct value, so if they don't agree, no value is written which masks the value fault.

### Graph Grammar Representation

The graph grammar presentation illustrated in Figure 7.4 shows the context-sensitive production rule that transform a single *ft_ida* to an arrangement of parallel IDA's with multicaster and voter type components.

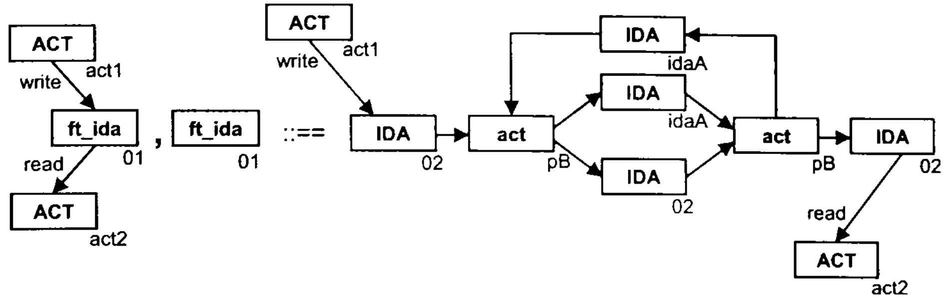



Figure 7.4: Template #4 - Temporal Redundant IDA

Additionally, a feedback IDA is shown which signals (like the watchdog template) whether both values received agreed.

### Fault Hypothesis

The specific faults, as specified in Section 4.2, for this template are:

$$
\begin{aligned}
&WriteValue(s.out_pB, i, t) \triangleq \\
&\quad \exists t' : Time \cdot t' \leq t \wedge \Theta(\uparrow s, i, t') \wedge \Theta(\downarrow s, i, t) \wedge \\
&\qquad\qquad pre_(s.in_pB(t'), \overleftarrow{v}(t')) \wedge \\
&\qquad\qquad \neg post_(s.in_pB(t'), \overleftarrow{v}(t'), v(t), s.out_pB(t))
\end{aligned}
$$

$$
\begin{aligned}
&OmitWrite(s.out_pB, i, t) \triangleq \\
&\quad \exists t', t'' : Time \cdot t' < t \leq t'' \wedge \Theta(\downarrow s, i, t') \wedge \\
&\qquad\qquad (\Theta(\uparrow s, i+1, t'') \vee \Theta(wds(s.out_pB), i+1, t'') \wedge \\
&\qquad\qquad OMIT(we(s.out_pB), i, [t', t''])
\end{aligned}
$$

$$
\begin{aligned}
&LateWrite(we_pB, i, t) \triangleq \\
&\quad \exists t' : Time \cdot t' + u1 < t \wedge \Theta(\uparrow s, i, t') \wedge \\
&\qquad\qquad LATE(we_pB, i, [t' + l1, t' + u1])
\end{aligned}
$$

### RTL

The specific property we propose this template offers is described in RTL as:

$$
\begin{aligned}
&\forall i : Occ, t_1 : Time \cdot \Theta(we_p1, i, t_1) \ \Rightarrow \\
&\qquad\qquad \exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(rd_p2, i, t_2)
\end{aligned}
$$

### 7.4.2 Fail-Signal (Activity)

The added functionality of this transformation is to mask certain failures to one expected type such that they may be corrected or tolerated by a further transformation or component. This template extends those

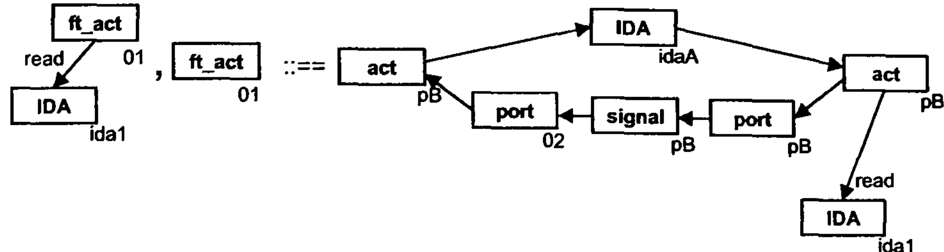



Figure 7.5: Template #5 - Fail Signal

presented previously, by using a feedback mechanism to detect the failures. The *tolerance* behaviour is to fail-silent whenever a fault is detached. This behaviour is desirable when spurious output of non-timely data is harmful to a system overall performance. Whereas the watchdog timer template wrote a default output, this template should terminate *silently*.

## Graph Grammar Representation

The transformation proposed is illustrated in Figure 7.5. It is obvious to see the feedback mechanism signals to the *tolerant* activity if an output was successfully received. Given the absence of this signal then the activity should terminate, hence failing silently.

## Fault Hypothesis

The specification which defines this behaviour is that defined as a *crash* fault, defined in Section 2.1.3 as:

$$
\begin{aligned}
& Crash(act, i, t) \triangleq \\
& \quad \exists t' : Time \cdot t' < t \wedge \Theta(\upharpoonleft s, i, t') \wedge \\
& \qquad OMIT(\downharpoonleft s, i, [t' + s.bcet, t]) \wedge OMIT(\upharpoonleft s, i+1, [t' + s.bcet, t])
\end{aligned}
$$

### 7.4.3 Fail Stop (Activity)

A further variation to discuss is inspired by [SS83] who introduced fail-stop processes. The benefits for extending the TMR template to include fail-stop activities allows the overall tolerance of the TMR templates to accommodate a wider degree of faults. For example, if each activity were itself tolerant to timing faults, then the TMR template also to tolerant to value faults, then the overall tolerance is greater.

# Chapter 8

# Case Study - "A BVRAAM Launch System"

## Contents

**8.1 Identification** . . . . . 169
**8.2 Design overview** . . . . . 169
8.2.1 Abstract Design description . . . . . 172
**8.3 The Initial (Host) Design** . . . . . 173
8.3.1 Transformations . . . . . 175
8.3.2 RTN-SL Specification . . . . . 184
**8.4 A SHARD Analysis** . . . . . 184
8.4.1 Guide Words . . . . . 184
8.4.2 Analysis Results . . . . . 185
**8.5 Fault Hypotheses for the BVRAAM Launcher Design** . . . . . 190
8.5.1 RTL Specifications of $FH_i$ . . . . . 190
8.5.2 RTL Safety Theorems for $FH_i$ . . . . . 192
**8.6 The Transformed Design** . . . . . 193
8.6.1 Transformations . . . . . 193
**8.7 Axiomatic Semantics generated by $\Omega_f$** . . . . . 196
**8.8 Validation** . . . . . 196
**8.9 Evaluation** . . . . . 203

The aim of this Chapter is to illustrate each theoretical point presented earlier with a real-life system design, one which is common to the British Defence industry.

We first outline the motivation and reasons for choosing this case study which describes a software design fragment of a hypothetical Beyond Visual Range Air-to-Air Missile (BVRAAM) launch system. We summarise the SHARD analysis for the initial abstract design and draw our fault-hypotheses (FH) from the studies findings. We then show each stage of our design methodology to transform the initial design

| Mode Name | Mode Number |
|---|---|
| PUC | 1 |
| Pre-Launch | 2 |
| Abandon | 3 |
| Ready-to-Launch | 4 |
| Launch | 5 |
| Flight | 6 |
| Acquisition | 7 |
| Terminal | 8 |
| Abort | 9 |

Table 8.1: Mode Encoding

to one *more* tolerant to the faults identified and show justification of our claims that i) a graph grammar is a suitable design technique, ii) a transformation is *localised* (i.e. faults are a conservative extension) and finally iii) that this structured approach lends itself to structured formal verification.

## 8.1 Identification

The case study we have chosen describes aspects of a Beyond Visual Range Air-to-Air Missile (BVRAAM) in sufficient detail to enable our investigations. Although this study does not report on a specific product development, it is hoped that any design engineering techniques which are applicable to the case study presented here will also be applicable to real products.

The benefit of choosing this example has been its use previously to develop safety & design engineering techniques [Pay01c, Pay01b, Pay01a] in a collaborative study. This study raised a number of interesting challenges [Pay01a] for anyone wishing to apply formal techniques to the specification, validation or verification of RTS.

## 8.2 Design overview

The design is based around a number of modes the system can operate within. Dependant on the modes are he behaviours and reaction a system can generate. The possible modes and the transitions between them are specified in Table 8.2, where the modes are numbered as defined in Table 8.1.

### 8.2.1 Abstract Design description

This section contains an informal description of each of the activities identified in Figure 8.1, the reader is referred to [Pay01c] for a fuller description. This level of detail is a common starting point for real-time systems.

**Mode Controller,** $MC_a$ The *Mode Controller* activity loops, reading "events" from the *Mode Events*, $ME_i$ channel to determine the next system mode to enter, given the current mode and the event read. The

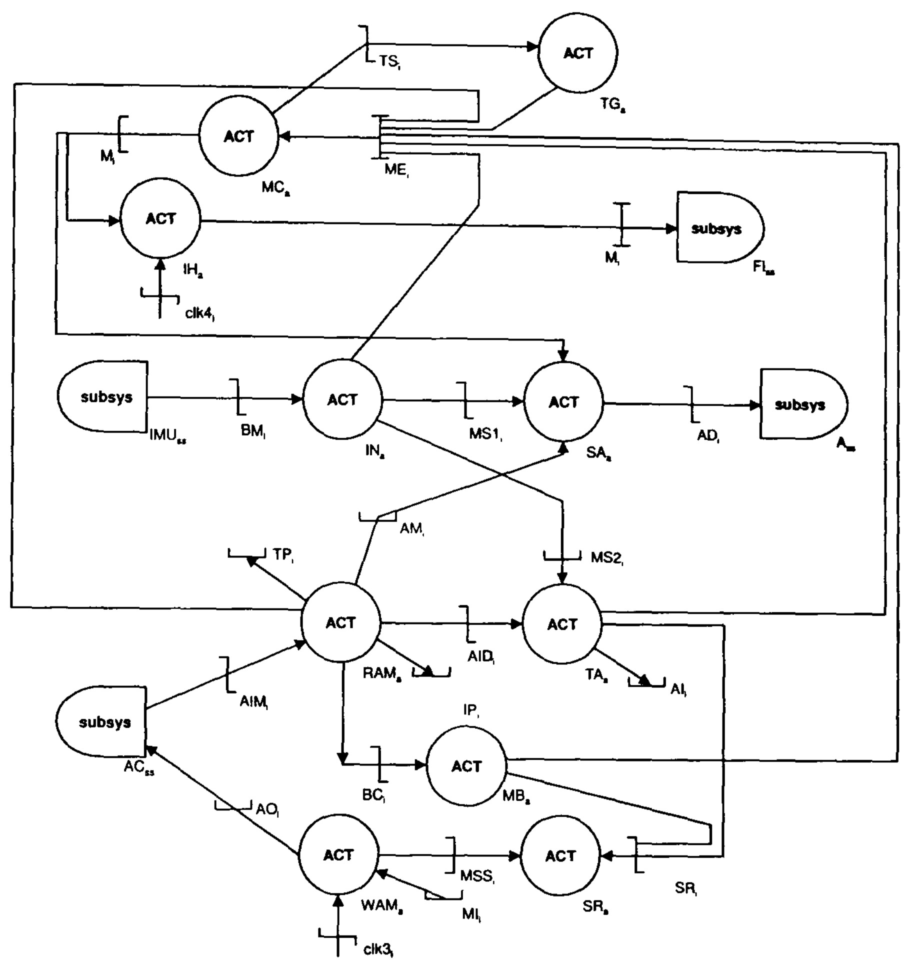



Figure 8.1: A BVRAAM Launcher Design

| Event \ Current Mode | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| BIT ok | 2 | - | - | - | - | - | - | - | - |
| BIT fails | 3 | - | - | - | - | - | - | - | - |
| Cross-Check Fails | - | 3 | - | - | - | - | - | - | - |
| Cross-Check ok | - | 4 | - | - | - | - | - | - | - |
| Umbilical cut | - | - | - | 5 | - | - | - | - | - |
| Clear (a timeout) | - | - | - | - | 6 | - | - | - | - |
| Acquired | - | - | - | - | - | 7 | - | - | - |
| Arrived | - | - | - | - | - | - | 8 | - | - |
| Fuzing Timeout | - | - | - | - | - | - | - | 9 | - |

Table 8.2: Previous/Next Mode Table

deterministic choice of which mode transition to taken is shown in Table 8.2. The new mode is written back into the $M_i$ pool. The activity then begins the loops again. As the activity reads from the $ME_i$ channel, if there are no new events to process the activity is *held* until an event occurs, which is processed as specified in Table 8.2.

**Inertial Navigation,** $IN_a$ The *Inertial Navigation,* $IN_a$ activity loops, attempting to read the periodically generated *Body Motion,* $BM_i$ data from the missile *IMU,* $IMU_{ss}$ subsystem. The $IN_a$ activity is responsible for generating the integrity mode events, written to $ME_i$. Should no check fail, then the $IN_a$ activity writes the body motion data of the launch vehicle to the separation autopilot, and via a pool to the transfer alignment activity. Effectively, this activity loops at the frequency of the $IMU_{ss}$ subsystem, determined by the IMU clock, $clk1$.

**Seperation Autopilot,** $SA_a$ The Separation Autopilot, $SA_a$ activity loops, reading the *Missile State,* $MS1_i$ data from the initial navigation activity$IN_a$. The current mode then determines whether initialisation or launch sequence data is calculated for the Actuators, $A_{ss}$ subsystem. Otherwise the activity loops until such functionality is required.

**Transfer Alignment,** $TA_a$ The Transfer Alignment, $TA_a$ activity loops, comparing the data from the *Aircraft INS Data,* $AID_i$ signal and the *Missile State,* $MS_i$ pool to ensure the IMU is operating within some defined *tolerance* which raises either a "Cross-Check" pass or fail accordingly.

**Read Aircraft Messages,** $RAM_a$ The Read Aircraft Messages, $RAM_a$ activity loops, writing incoming target position, missile initial position and aircraft INA data to the *Target Position,* $TP_i$ pool, *Initial Position,* $IP_i$ pool and *Aircraft INS Data,* $AID_1$ signal respectively, until the *Umbilical Cut* event is detected, which is raised to the $ME_i$ channel.

**Write Aircraft Messages,** $WAM_a$ The Write Aircraft Messages, $WAM_a$ activity loops, reading the stim event from clock, $clk3$. It reads the *Missile Identity,* $MI_i$ and *Missile Status Summary,* $MSS_i$ pools and writes this information to the $AO_i$ pool.

**Interlock Handler,** $IH_a$ The Interlock Handler, $IH_a$ activity loops, waiting for a tick from clock, $clk4$. It then checks the current mode, and if it is "Abandon", it sets the firing interlock in an attempt to prevent the launch.

**Status Reporting,** $SR_a$ The Status Reporting, $SR_a$ activity loops, reading status reports from the *Status Report,* $SR_i$ signal. It writes a compiled report to the $MSS_i$ pool.

**Manage BIT,** $MB_a$ The Manage BIT, $MB_a$ activity loops, awaiting BIT commands from *BIT Command,* $BC_i$ signal. The activity then performs a series of built-in-tests and reports the success or failure to $SR_i$ and $ME_i$.

**Timeout Generator,** $TG_a$ The Timeout Generator, $TG_a$ activity loops, waiting for timeout requests from the $MC_a$. Upon being reactivated, the activity sends the timeout (clear) events to $ME_i$.

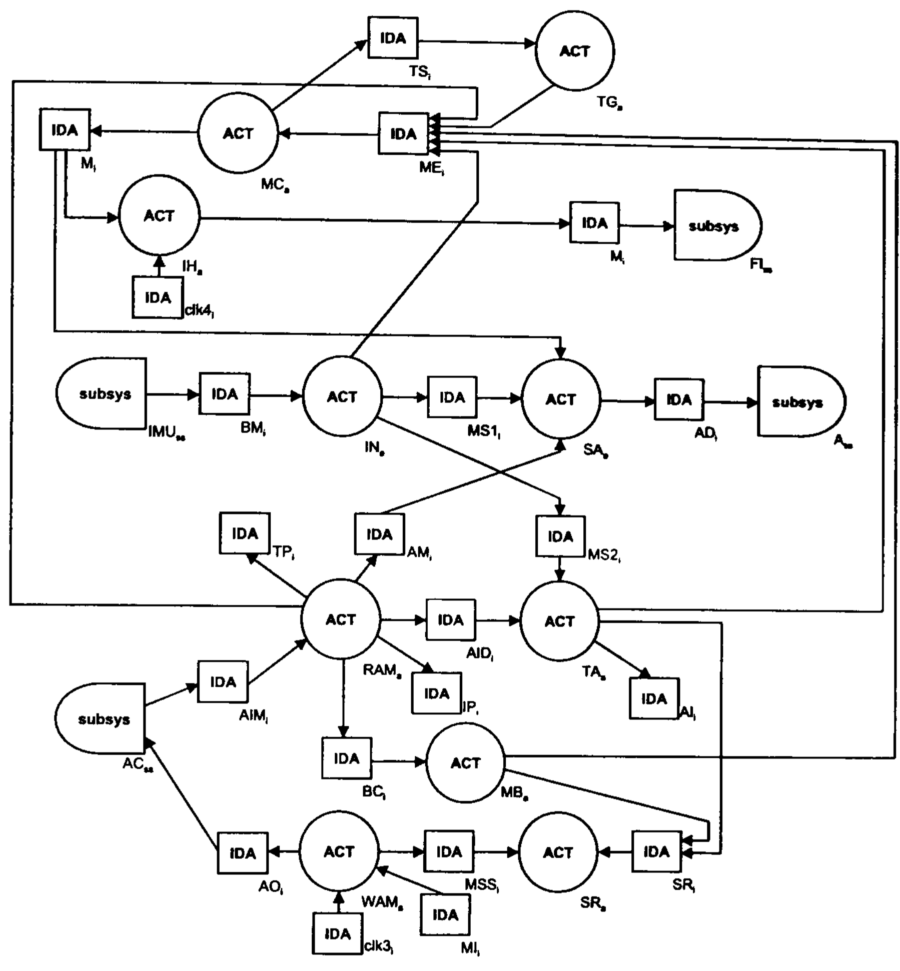



Figure 8.2: Host Graph

## 8.3 The Initial (Host) Design

The initial MASCOT design taken from [Pay01c] illustrated in Figure 8.1 serves as the basis for our case study. This design specifies only the network layer: identifying those activities (described previously) and communication paths between activities necessary. Some design decisions have already been made, choosing the protocols of each IDA specifies the synchronisation between the activity components.

In the next section we identify the graph grammar representation of the initial design which is termed our *host graph*. From this, we illustrate the 24-rule grammar adequate to derive the application layer for the RTN. That is, to refine the abstract activity description into state-machines which read and write to the identified communication paths. We then present the RTN-SL specification of the transformed host graph, which specifies the RTN behaviour without faults being considered.

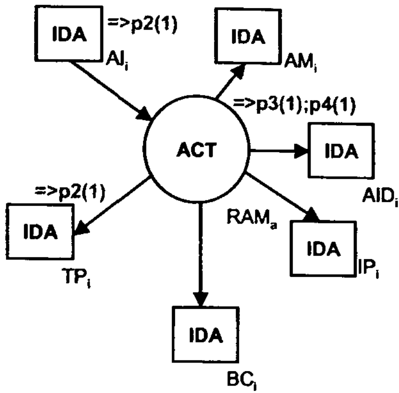



Figure 8.3: Neighbourhood of $RAM_a$

### 8.3.1 Transformations

The (host) graph depicted in Figure 8.2 is the graph grammar abstract representation of the initial design from Figure 8.1. Below we show as examples, the derivations of activity (ACT) nodes to their state-machine representation.

#### 8.3.1.1 Refining the *Read Aircraft Messages,* ($RAM_a$) Activity

We present in Figures 8.3–8.9 a derivation from the ACT node in Figure 8.2 which represented $RAM_a$. We follow a series of transformation steps to refine the activity first to a *state-machine* (SM) which implements the desired behaviour, as specified in the activity descriptions in Section 8.2.1.

The *neighbourhood* of the node labelled $RAM_a$ is shown in Figure 8.3. Incident to the node being transformed are six IDA components which form the neighbourhood of the node. The connections from the existing neighbourhood nodes determine the new connection as specified by the embedding relations of each production rule.

The transformation from Figure 8.3 to Figure 8.4 is by way of several steps. Nodes $AI_i$ and $TP_i$ are both transformed by production rule[1] *net_p2(1)*.The other transformation *net_p3(1);net_p4(1)* to node $RAM_a$ indicates rule *net_p3(1)* is first applied, then *net_p4(1)* to its result below. This transforms the ACT node to the most abstract state-machine representation. Note, each communication link which was established to the ACT node remains connected to the SM node.

The applied transformation from Figure 8.4 to Figure 8.5 depicts the non context-sensitive state-machine production rule, *net_p5(3)*. Here, the incoming data communication from node $AI_{opt}$ to re-establish to both node $s_1$ and $RAM_{SM}$ indicating the port may be read by more than one dynamic state (later we shall see the context sensitive counterpart where a incoming connection is enforced to connect to only one node).

The transformation from Figure 8.5 through to Figure 8.8 further refines the state-machine specification. However, the final transformation illustrated from Figure 8.8 to Figure 8.9 shows an example of a context-

[1]The bracketed number indicates the sub rule of a family of production rules.

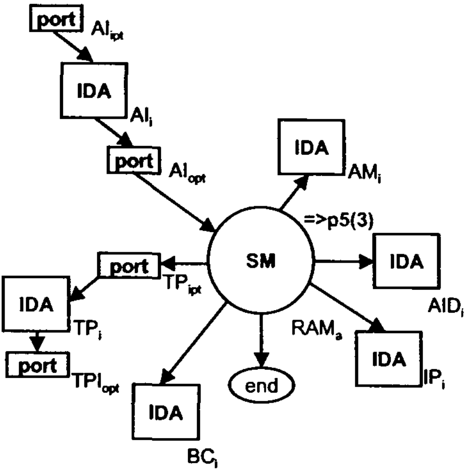



Figure 8.4: $RAM_a$ - p1

sensitive production rule being applied. The transformation *net_p6(b)* applied to node $SM_1$ in Figure 8.8 transforms a state-machine with multiple port (and IDA) connections to a single state node which now only has one connection, that to an associated port. Existing state-machine transitions are restored by the embedding relation.

The remaining transformation are not shown here in such detail, the reader is referred to the RTN-SL specification for the complete specification of the $RAM_a$ activity.

#### 8.3.1.2 Refining the *Transfer Alignment*, ($TA_a$) Activity

Similarly, we present below a derivation from the ACT node in Figure 8.2 which represented the activity node for $TA_a$.

The series of transformations from the abstract ACT node, $TA_a$ to its (flat) state-machine definition is detailed in Figures 8.10 through Figure 8.14. Similarly to the transformation of $RAM_a$, we gradually refine the state-machine specification to a collection of states which read from, and write to, specific ports and IDAs. The transformation between Figures 8.12–8.13 illustrates another example of a context sensitive production rule, as does the transformation between Figures 8.13–8.14.

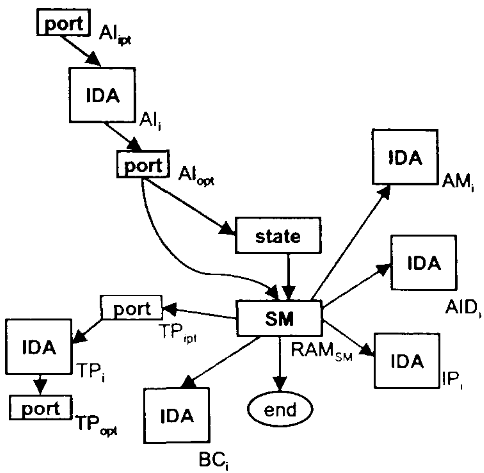



Figure 8.5: $RAM_a$ - p2

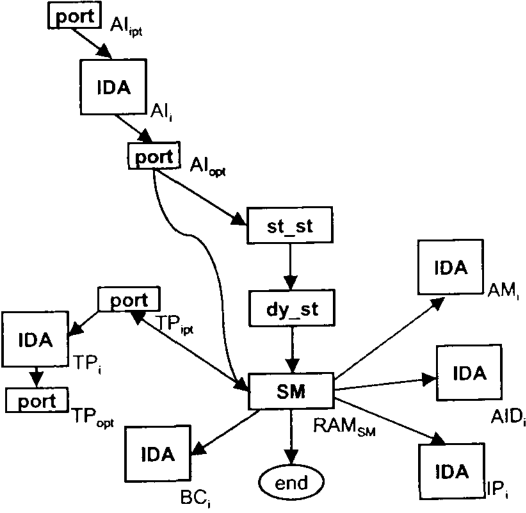



Figure 8.6: $RAM_a$ - p3

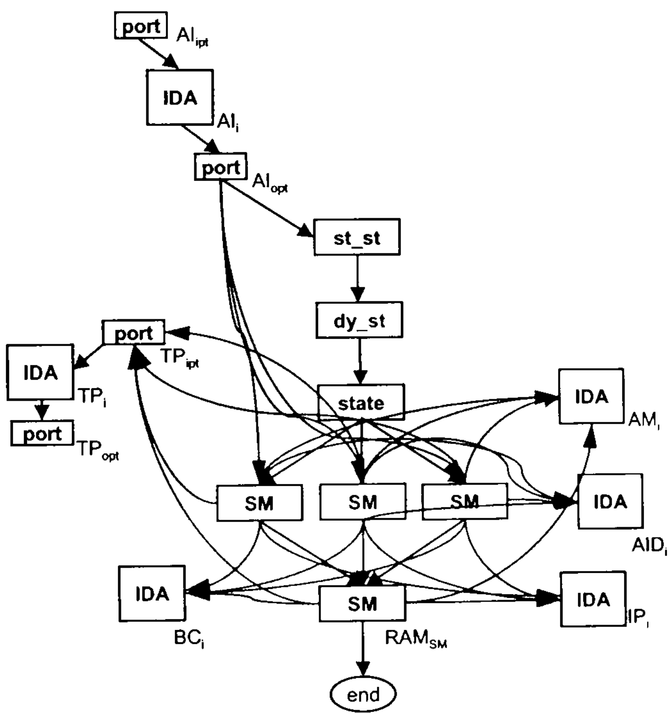



Figure 8.7: $RAM_a$ - p4

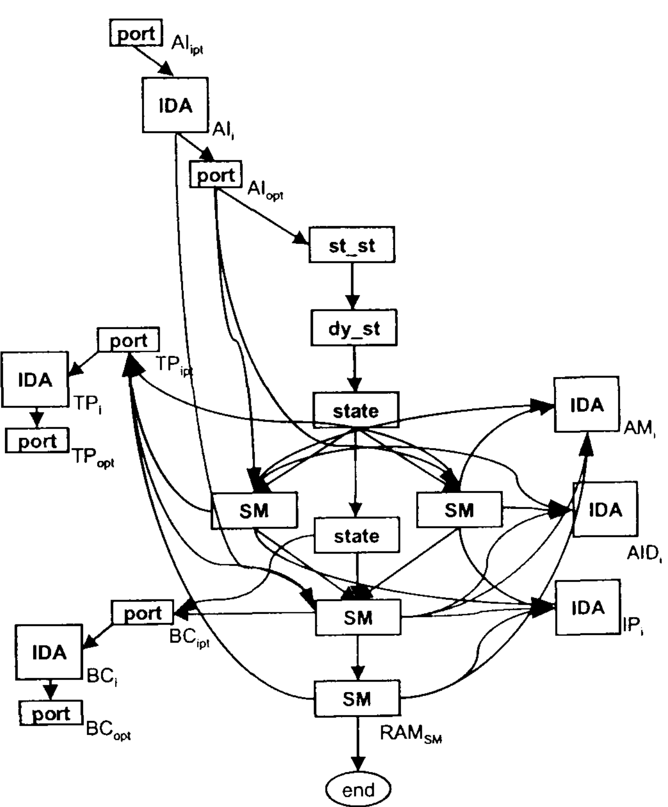



Figure 8.8: $RAM_a$ - p5

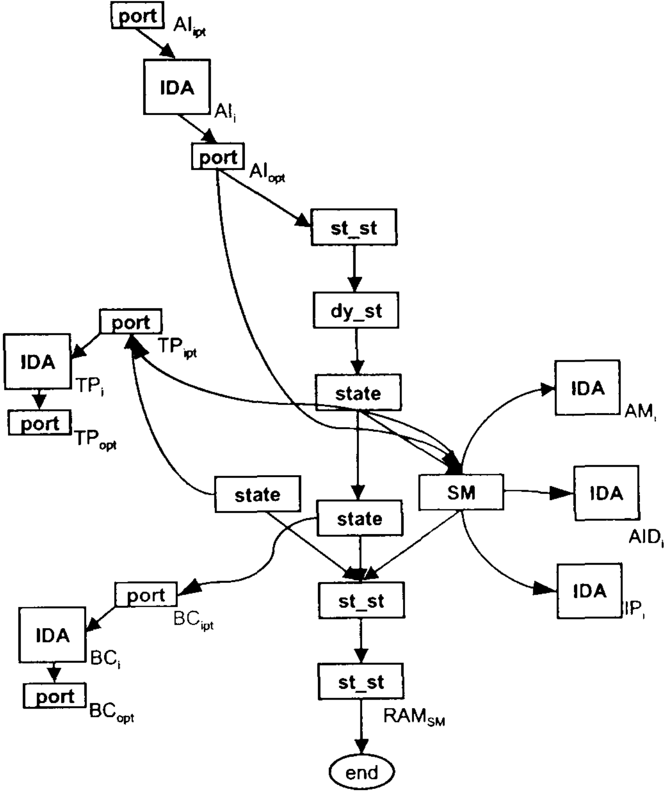



Figure 8.9: $RAM_a$ - p6

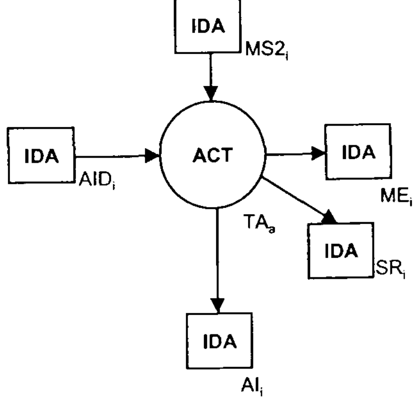



Figure 8.10: Neighbourhood of $TA_a$

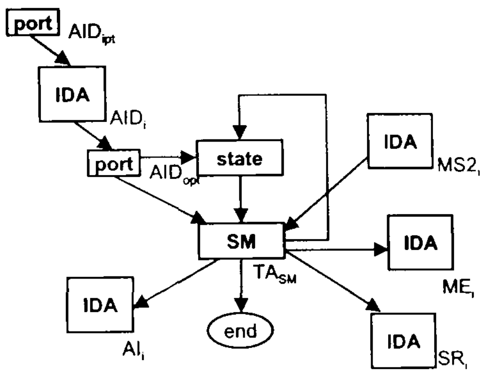



Figure 8.11: $TA_a$ - p1

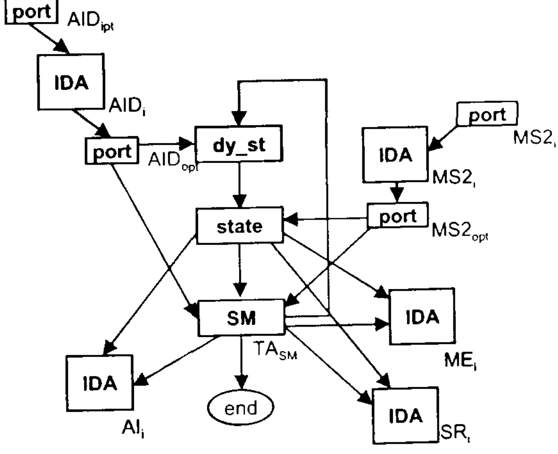



Figure 8.12: $TA_a$ - p2

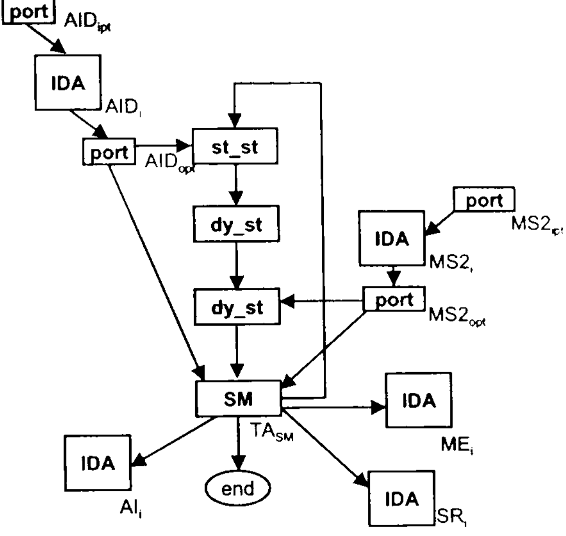



Figure 8.13: $TA_a$ - p3

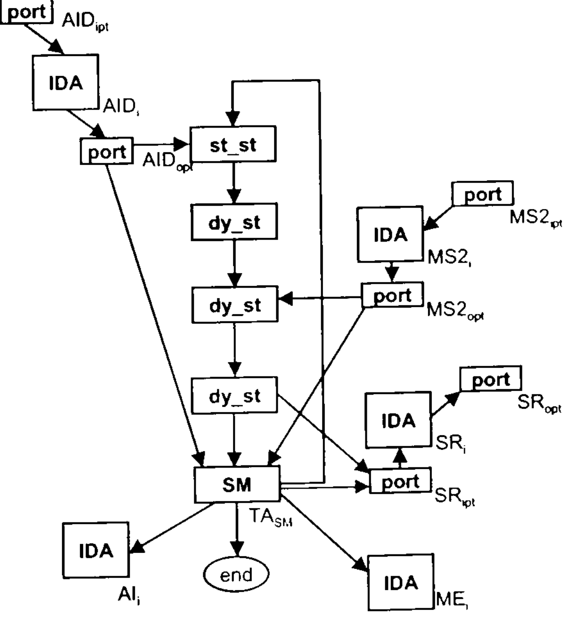



Figure 8.14: $TA_a$ - p4

| | Failure Categorisation | | | | | |
|---|---|---|---|---|---|---|
| | Service Provision | | Timing | | Value | |
| Component | Omission | Commission | Early | Late | Subtle | Coarse |
| Pool | No Update | Unwanted Update | N/A | Old Data | Incorrect | N/A |
| Signal | No Data | Extra Data | Early | Late | Incorrect | Inconsistent |
| Channel | No Data | Extra Data | Early | Late | Incorrect in Range | Out of Range |
| Dynamic State | No Update | Unwanted Update | Early Exit | Late Exit | Incorrect in Range | Out of Range |
| Static State | N/A | N/A | Early Exit | Late Exit | N/A | N/A |
| Activity | Crash | N/A | Early | Late | N/A | N/A |

Table 8.3: Table of guide words applicable to RTNs

### 8.3.2 RTN-SL Specification

The design specification that is found in Appendix A.1 is a complete RTN-SL specification of the initial design introduced previously. It defines in greater detail each activities behaviour and its communication paths between sub-systems and other activities. The IDA specifications define the communication protocol and their synchronous behaviour.

Only those components we foresee to be of interest to our methodology are fully specified, the remaining components are simplified so the RTN-SL design is well-formed; this simplification is normally a trivial post-condition on operations. However, the complete RTN-SL specification is of sufficient detail to generate the event model required for the formal reasoning.

## 8.4 A SHARD Analysis

We now report the findings from a SHARD analysis of our initial design, as advocated in Section 2.5.1. We first repeat the guide words applicable for such an example, followed by selected presentation of the analysis. We aim to show the obvious benefit of such an analysis and draw our fault hypothesis (FH) from which we apply our technique.

### 8.4.1 Guide Words

The guide words in Table 8.3 (repeated from Table 2.3) are the least set which we are confident should prompt consideration of the plausible failure modes for RTNs.

### 8.4.2 Analysis Results

Consider the design of Figure 8.1 and the permutations of guide words (from Table 8.3) to each *data flow* component of our design, then it is obvious a full SHARD analysis is not appropriate here. We are not advocating a new safety analysis technique, rather one which given scope for possible failures, addresses each in a systematic approach to make the design more tolerant. However, given the SHARD analysis is

a valuable tool, we therefore undertake such an analysis into key areas of the design where the abstract descriptions hint at possible failure modes to tease out the specific faults and their failure modes.

We present here only a fragment of the tabular result set which are most appropriate. Each table that follows has the following format:

- A name & reference for the drawing/design being analysed;
- the data flow Id, protocol & type;
- Any additional information deemed necessary;
- For each guide word in Table 8.3, we
  - identify any deviation from the specification the guide word prompts,
  - its possible causes and detection mechanisms
  - any co-effectors
  - the **observed** effects
  - whether the fault is meaningful, which determines whether we address the fault in our technique
  - the SHARD justification for not considering it meaningful or the desired action necessary for any tolerance technique.

| | |
|---|---|
| **Drawing Ref** | Host-Design (c.f. Figure 8.1) |
| **Drawing Name** | Top Level Design |
| **Flow ID** | $AIM_i$ |
| **Protocol** | Signal |
| **Data Type** | Image.Raw_Image |
| **Additional Information** | This is the input feed from the Aircraft sub-system. In-coming messages from the aircraft will be propagated along different paths depending on their type. |

| Guide Word | Deviation | Possible Causes | Detection / Protection | Co-effectors | Effects | M? | Justification / Design Proposals |
|---|---|---|---|---|---|---|---|
| Late | No new image sent to $RAM_a$ | Late read @ $p13$ | Undetectable | | Delayed updated to rest of system | **N** | **Justification:** By design, $RAM_a$ is designed to wait for the signal from $AIM_i$ |
| Value, coarse | Impossible data from $A_{ss}$ | Communication corrupted | CRC ensures corrupted messages fail silent | | Lost Communication | **N** | **Justification:** Due to feedback loops, $A_{ss}$ will attempt to resend |

Table 8.4: SHARD analysis of flow $AIM_i$ of the BVRAAM Launch system top level design

**Drawing Ref** Host-Design (c.f. Figure 8.1)
**Drawing Name** Top Level Design
**Flow ID** $MS2_i$
**Protocol** Pool
**Data Type** Missile_State.IMU_Data
**Additional Information** This is the information calculated from the $IN_a$ activity which calculates the body motion of the aircraft which is sent to the $TA_a$ activity via $MS2_i$.

| Guide Word | Deviation | Possible Causes | Detection / Protection | Co-effectors | Effects | M? | Justification / Design Proposals |
|---|---|---|---|---|---|---|---|
| Value, subtle | Comparison with $p20$ fails | Byzantine fault @ $p20$ or $p21$; Transmission error; communication error | Comparison check fails | Failure @ $RAM_a$ or $IN_a$ | Non-deterministic state | Y | **Action:** Make fault tolerant |
| No Update | No IMU Data sent to $TA_a$ | Communication failure; $IN_a$ failure | Comparison check fails | | Old data used in comparison check | Y | **Action:** Make fault tolerant |

Table 8.5: SHARD analysis of flow $MS2_i$ of the BVRAAM Launch system top level design

**Drawing Ref** Host-Design (c.f. Figure 8.1)
**Drawing Name** Top Level Design
**Flow ID** $MS1_i$
**Protocol** Signal
**Data Type** Missile_State.status
**Additional Information** This is the information calculated from the $IN_a$ activity which calculates the body motion of the aircraft which is sent to the $SA_a$ activity via $MS1_i$.

| Guide Word | Deviation | Possible Causes | Detection / Protection | Co-effectors | Effects | M? | Justification / Design Proposals |
|---|---|---|---|---|---|---|---|
| Value, Coarse | Incorrect status sent | Byzantine Fault | | $SA_a$ & $TA_a$ | Inconsistent internal state of system | Y | **Action:** Make fault tolerant |
| Omission | No signal sent | $IN_a$ fails to write signal | | | Latency | N | **Justification:** By design |
| Late, write | | Delay to $IN_a$ | | | Latency | Y | **Action:** Make fault tolerant |

Table 8.6: SHARD analysis of flow $MS1_i$ of the BVRAAM Launch system top level design

**Drawing Ref** Host-Design (c.f. Figure 8.1)
**Drawing Name** Top Level Design
**Flow ID** $BM_i$
**Protocol** Signal
**Data Type** Image.Processed_Image
**Additional Information** This is the body motion data of the aircraft sent periodically from the aircraft, $A_{ss}$ sub-system.

| Guide Word | Deviation | Possible Causes | Detection / Protection | Co-effectors | Effects | M? | Justification / Design Proposals |
|---|---|---|---|---|---|---|---|
| Extra Data | Unexpected arrival | Period of $clk1$ | | | Missed Data | Y | **Action:** Reconfigure $clk1$ |
| Value, subtle | Repeated data | Stuck at Value | **Comparisons** with local state | $SA_a$ & $TA_a$ | Events raised from unconsidered modes | Y | **Action:** Make fault tolerant |

Table 8.7: SHARD analysis of flow $MB_i$ of the BVRAAM Launch system top level design

## 8.5 Fault Hypotheses for the BVRAAM Launcher Design

We now interpret the SHARD analysis to form our own fault hypothesis, FH. We have identified five meaningful faults which we consider should be addressed in response to the SHARD analysis to continue to study the initial safety analysis given the admissions of faults. These faults are listed below:

**FH1:** The alignment between the missile INS data and that of the launch aircraft may exceed some *tolerance* value; and

**FH2:** The actuator commands may not satisfy the specified post-condition; and

**FH3:** The "Abandon" command may not be actioned within the specified deadline; and

**FH4:** The ordering of events read by the $MC_a$ from the $ME_i$ IDA may not be processed in the actual order they were sent; and finally

**FH5:** The $SA_a$ activity may generate actuator commands late, i.e. beyond the specified deadline.

We will refer in future to these fault hypotheses as **FH1, FH2, ..., FH5**.

### 8.5.1 RTL Specifications of $FH_i$

For each $FH_i$ identified, we now give RTL specifications which are the actual specifications from Section 4.2.1 and the initial design. Theses specifications will be used in the formal reasoning of the eventual transformed design

#### FH1

The "Transfer Alignment" ($TA_a$) activity receives a vector (*Missile_State.pos*) from the missile IMU and a second vector (*Image.Vector*) from the aircraft, both of the same type (*Image.Vector*). A vector: $(x, y, z)$ which gives the objects position in a 3-D space. Given the timing of receipt, the two positions may not agree for two principal reasons: 1) the vectors were 'valued' at different times (the aircraft and missile IMU will certainly work at different periods) or 2) the missile position on the aircraft 'offsets' the position. In both cases, a degree of *tolerance* between the values is acceptable, as long as the difference remains constant, or doesn't **drift**. The function which calculates the comparison is *Missile_State.compare*:

```
function compare (IMU, AC, offset : Image.Vector)
          return res : bool;
  pre true;
  post align(IMU, offset) = AC;
```

**fault_thm1:**

$$\exists i : Occ, t_1, t_2 : Time \cdot \Theta(\uparrow TA_C, i, t_1) \wedge \Theta(\downarrow TA_C, i, t_2)$$
$$\Rightarrow \neg post_TA_Check_Missile_Alignment(TA_incoming_ms(t_1)), TA_input(t_1))(TA_MS(t_2))$$

FH1 permits the *post*-condition of *Missile_State.compare* not to hold at some time, $t_2$, which in turn contradicts the exit transition from *TA_C*. We therefore must tolerate this fault, such that a transient fault –possibly due to the period of inputs– does not affect the system behaviour.

**FH2 and FH5**

The "Separation Autopilot" ($SA_a$) activity must produce updated actuator commands to $p10$ from the input at $p9$ in both a timely and value tolerant fashion. On receiving updated missile IMU data on $p9$, the $SA_a$ activity calculates updated actuator commands whilst in flight to prevent spinning.

The $SA_a$ activity receives input from the Inertial Navigation system mode, from the Mode ($M_i$) IDA. Given the system mode may be "Launch", the actuator commands received from the Inertial Navigation activity –to prevent the launch craft being hit– are calculated to be the launch commands for the actuators. This sequence of events must be timely, else the inertial commands become *dated*, such as the launch vehicle changing course.

**fault_thm2:**

Value: $\exists i : Occ, t_1, t_2 : Time \cdot \Theta(rd_p9, i, t_1)$
$\Rightarrow \Theta(we_p10, it_2) \wedge \neg post_SA_launch(SA_MS(t_1), SA_CM(t_1))(SA_launch_cmd(t_2)) \wedge t_2 \leq t_1 + u$

**fault_thm5:**

Time: $\exists i : Occ, t_1, t_2 : Time \cdot \Theta(rd_p9, i, t_1)$
$\Rightarrow \Theta(we_p10, i, t_2) \wedge post_SA_launch(SA_MS(t_1), SA_CM(t_1))(SA_launch_cmd(t_2)) \wedge t_2 \geq t_1 + u$

**FH3**

The "Interlock Handler" ($IH_a$) activity must activate the firing lock upon receiving the "Abandon" command. The abandon mode would be received via the Mode ($M_i$) IDA, read at port *p3*. Given the implications of this request not being performed, it is necessary to guarantee the firing lock is activated timely upon its request.

**fault_thm3:**

$$\exists i : Occ, t_1 : Time \cdot \Theta(rd_p3, i, t_1) \wedge post_(abandon, i, t_1) \Rightarrow \exists j : Occ, t_2 : Time \cdot t_1 + X < t_2 \wedge \Theta(we_p4, j, t_2)$$

**FH4**

The nature of this fault hypothesis prohibits an RTL specification as $FH_1, FH_2, FH_3 \& FH_5$ have provided. Instead, the SHARD investigation has highlighted a necessary behavioural characteristic. The ***design fault*** to this hypothesis is that a multi-writer channel has been decided upon that can prevent crucial messages being read promptly. For example, non-critical messages may be before a "Abandon" command –for example, the "Launch" command– in this FIFO data structure.

### 8.5.2 RTL Safety Theorems for $FH_i$

Given the specification of the faults admissible to our design, we must now state the safety theorems we must prove to hold for the transformed design.

**safety_thm1:**

$$\forall i : Occ, t_1 : Time \cdot \Theta(rd_p21, i, t_1) \Rightarrow \Theta(we_p23, i, t_1)$$

**safety_thm2:**

$$\forall i : Occ, t_1, t_2 : Time \cdot \Theta(rd_p9, i, t_1) \wedge \Theta(we_p10, i, t_2) \Rightarrow post_(p9(t_1), p10(t_1))$$

**safety_thm3:**

$$\forall i : Occ, t_1 : Time \cdot \Theta(rd_p3, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(we_p4, i, t_2)$$

**safety_thm5:**

$$\forall i : Occ, t_1 : Time \cdot \Theta(we_p5, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(we_p23, i, t_1)$$

## 8.6 The Transformed Design

### 8.6.1 Transformations

We now present the graph grammar transformations to the host graph which tolerate each fault hypothesis.

#### 8.6.1.1 Addressing $FH_1$

The transformation illustrated in Figure 8.15 shows the PSR template applied to provide tolerance to $FH_1$. From Figure 8.14, we have applied the PSR template described in Chapter 7, to node $ds_3$. The resulting transformation now provides for a degree of tolerance to the output produced to $p23$.

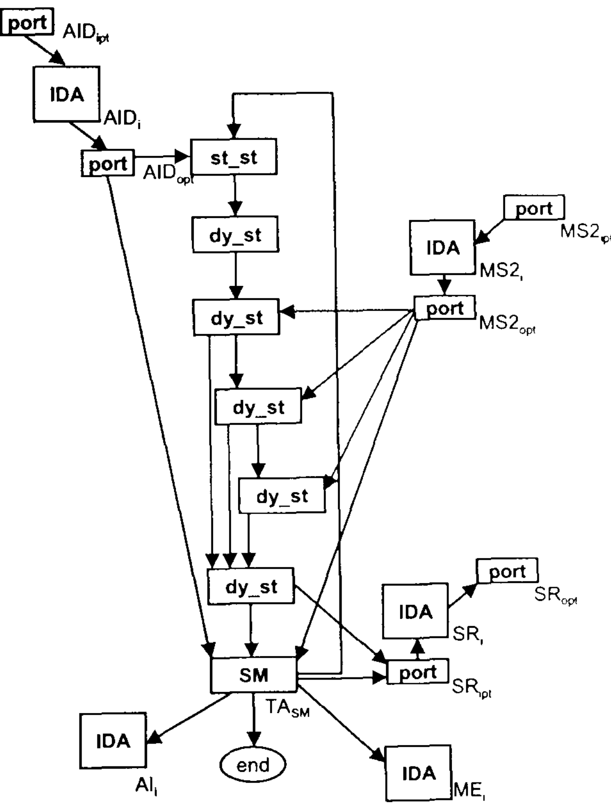



Figure 8.15: $TA_a$ - pFT1

#### 8.6.1.2 Addressing $FH_2$ and $FH_5$

Given the same transformation tolerates both the faults defined by $FH_2$ and $FH_5$, we will present the transformations as one. We propose to apply Template #2 from Chapter 7 to node $SA_a$ in our host design. Although the precise context to $SA_a$ is not that shown in Figure 7.2, it does follow the structure discussed in Section 7.2.4. The precise characteristics of the design of $SA_a$ allow for the transformation are that there exists only one output, therefore only one output needs exist from the transformation.

Consider the other inputs to $SA_a$ would necessitate further multicaster components to rely the inputs to each replica. Given no faults are considered for these communication links, and the SHARD analysis did not indicate any potential faults from those components, we abstract the transformation to illustrate the transformation and reasoning for the given fault hypothesis.

#### 8.6.1.3 Addressing $FH_3$

We are required to guarantee the firing lock is in place in a timely manner. Therefore, allowing for the possibility of delayed inputs at $p3$, we propose to apply a watchdog timer strategy which guarantees to write a default output if no input s received at $p3$ for some lengthly period.

#### 8.6.1.4 Addressing $FH_4$

The fourth fault hypothesis found by the SHARD analysis identified that critical message to the multi-writer/reader at IDA $ME_i$ could be blocked (or held-up) by non-critical messages given the FIFO behaviour characteristics of a channel. It is therefore proposed to replace –or transform– this IDA component to that of an activity and in addition, replace the critical data path with the transformation presented as Template #4 in Chapter 7.

$FH_4$ is therefore addressed in two stages: i) the $ME_i$ IDA is replaced by a simple activity which reads from each producer to $ME_i$; ii) the communication path between this new activity is transformed as illustrated in Template #4.

## 8.7 Axiomatic Semantics generated by $\Omega_f$

We illustrate in Appendix A.2 the modified semantics generated by the aforementioned transformations. The axioms in Appendix A.2 which are then used in the rigorous proofs that follow to show the safety theorems listed above, which are shown to be correct under the assumption no faults occur remain true under the modified semantics.

## 8.8 Validation

PROOF (SAFETY_THM1) We first prove the safety theorem holds under the assumption no faults can occur.

from $Assumptions, ActivityTheorems, LinkAxioms, NOFaults$

| | | |
|---|---|---|
| 1 | from $i : Occ, t_1 : Time, \Theta(rd_p21, i, t_1)$ | |
| 1.1 | $\Theta(\uparrow TA_C, j, t_1)$ | TA_rds_p21_ax(1.h3) |
| 1.2 | $\Theta(\downarrow TA_C, j, t_3) \wedge t_3 > t_1$ | TA_ax_17(1.1) |
| 1.3 | $post_TA_check_missile_alignment()$ | TA_ax_31(1.1,1.2) |
| 1.4 | $\Theta(\uparrow TA_D, j, t_3)$ | TA_ax_5(1.2) |
| 1.5 | $\Theta(\downarrow TA_D, j, t_4) \wedge t_4 > t_3$ | TA_ax_18(1.4) |
| 1.6 | $\Theta(we_p23, i, t_4) \wedge t_4 > t_3$ | TA_we_p28_ax(1.5) |
| | infer $\exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | ∃-I(1.6) |
| 2 | $\Theta(rd_p21, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | ⇒-I(1) |
| | infer $\forall i : Occ, t_1 : Time \cdot \Theta(rd_p21, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | ∀-I(2) |

■

PROOF (FT_SAFETY_THM1) We then prove the safety theorem holds in the presence of the hypothesised faults.

from $Assumptions, ActivityTheorems, LinkAxioms, FH_1$

| | | |
|---|---|---|
| 1 | from $i: Occ, t_1: Time, \Theta(\Downarrow TA_B, i, t_1), FH_1$ | |
| 1.1 | $\Theta(\Uparrow TA_rp1, i, t_1)$ | TA_ax_4_rp1(1.h3) |
| 1.3 | $\Theta(rds_p21, i, t_1) \Leftrightarrow \exists j: Occ \cdot \Theta(\Uparrow TA_rp1, j, t_1) \vee \Theta(\Uparrow TA_rp2, j, t_1) \vee \Theta(\Uparrow TA_rp3, j, t_1)$ | TA_rds_p21_ax(1.1) |
| 1.3 | $\Theta(rds_p21, i, t_1)$ | $\Leftrightarrow$ -E |
| 1.4 | $\Theta(\Downarrow TA_rp1, i, t_3)$ | TA_ax_17_rp1(1.1) |
| 1.5 | $\neg post_TA_check() \vee post_TA_check()$ | TA_ax_31_rp1(1.1,1.4) |
| 1.6 | from $\neg post_TA_check()$ | |
| 1.6.1 | $\Theta(\Uparrow TA_rp2, j, t_3)$ | TA_ax_5_rp1 |
| 1.6.2 | $\Theta(rds_p21, k, t_3) \Leftrightarrow \exists j: Occ \cdot \Theta(\Uparrow TA_rp1, j, t_3) \vee \Theta(\Uparrow TA_rp2, j, t_3) \vee \Theta(\Uparrow TA_rp3, j, t_3)$ | TA_rds_p21_ax |
| 1.6.3 | $\Theta(rds_p21, k, t_3)$ | $\Leftrightarrow$ -right-E(1.6.1) |
| 1.6.4 | $\Theta(\Downarrow TA_rp2, k, t_4) \wedge t_4 > t_3$ | TA_ax_17_rp2(1.6.1) |
| 1.6.5 | $\neg post_TA_check() \vee post_TA_check()$ | TA_ax_31_rp2(1.6.1,1.6.4) |
| 1.6.6 | from $\neg post_TA_check()$ | |
| | same structure as above for rp3 | |
| | infer $\Theta(we_p23, i, t_5) \wedge t_5 > t_4$ | |
| 1.6.7 | from $post_TA_check()$ | |
| | same structure as above for rp3 | |
| | infer $\Theta(we_p23, i, t_5) \wedge t_5 > t_4$ | |
| | infer $\Theta(we_p23, i, t_2) \wedge t_2 > t_1$ | $\vee$-E(1.6.5,1.6.6,1.6.7) |
| 1.7 | from $post_TA_check()$ | |
| | as above | |
| | infer $\Theta(we_p23, i, t_2) \wedge t_2 > t_1$ | |
| 1.8 | | $\vee$-E(1.5,1.6,1.7) |
| | infer $\exists t_2: Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | $\exists$-I(1.h1,1.h2,1.h3) |
| 2 | $\Theta(\Downarrow TA_B, i, t_1) \wedge FH_1 \Rightarrow \exists t_2: Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | $\Rightarrow$-I(1) |
| | infer $\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow TA_B, i, t_1) \wedge FH_1 \Rightarrow \exists t_2: Time \cdot t_2 > t_1 \wedge \Theta(we_p23, i, t_2)$ | $\forall$-I(2) |

■

PROOF (SAFETY_THM2) We first prove the safety theorem holds under the assumption no faults can occur.

from $Assumptions, ActivityTheorems, LinkAxioms, NOFAULTS$

| | | |
|---|---|---|
| 1 | from $i : Occ, t_1, t_2 : Time, \Theta(rd_p9, i, t_1), \Theta(we_p10, i, t_2)$ | |
| 1.1 | $\Theta(\Downarrow SA_A, i, t_1)$ | SA_ax_19(1.h3,1.h4) |
| 1.2 | $\Theta(\Uparrow SA_Check_MS, i, t_1)$ | SA_ax_2(1.1) |
| 1.3 | $\Theta(\Downarrow SA_Check_MS, i, t_2) \wedge t_2 > t_1$ | SA_ax_14(1.2) |
| 1.4 | $\Theta(\Uparrow SA_Check_Mode, i, t_2)$ | SA_ax_3(1.3) |
| 1.5 | $\Theta(\Downarrow SA_Check_Mode, i, t_3) \wedge t_3 > t_2$ | SA_ax_15(1.4) |
| 1.6 | $\Theta(\Downarrow SA_Check_Mode, i, t_3) \Rightarrow SA_CM(t_3 \neq \text{launch} \wedge \Theta(\Uparrow SA_A, j+1, t_3) \vee$ $SA_CM(t_3 = \text{INIT} \wedge \Theta(\Uparrow SA_Init_act, j, t_3) \vee$ $SA_CM(t_3) = \text{launch} \wedge \Theta(\Uparrow SA_Calc_traj, i, t_3)$ | SA_ax_4(1.5) |
| 1.7 | from $SA_CM(t_3 \neq \text{launch} \wedge \Theta(\Uparrow SA_A, j+1, t_3)$ | |
| 1.7.1 | $SA_CM(t_3) \neq \text{launch}$ | ∧-E-right |
| | infer $post_launch(p9(t_1), p10(t_3))$ | contradiction |
| 1.8 | from $SA_CM(t_3 = \text{INIT} \wedge \Theta(\Uparrow SA_Init_act, j, t_3)$ | |
| 1.8.1 | $SA_CM(t_3 = \text{INIT}$ | ∧-E-right |
| | infer $post_launch(p9(t_1), p10(t_3))$ | contradiction |
| 1.9 | from $SA_CM(t_3) = \text{launch} \wedge \Theta(\Uparrow SA_Calc_traj, i, t_3)$ | |
| 1.9.1 | $\Theta(\Uparrow SA_calc_traj, j, t_3)$ | ∧-E-left |
| 1.9.2 | $\Theta(\Downarrow SA_calc_traj, j, t_4) \wedge t_4 > t_3$ | SA_ax_15(1.9.1) |
| 1.9.3 | $\Theta(\Uparrow SA_launch, j, t_4)$ | SA_ax_5(1.9.2) |
| 1.9.4 | $\Theta(\Downarrow SA_launch, j, t_5) \wedge t_5 > t_4$ | SA_ax_18(1.9.3) |
| | infer $post_launch(p9(t_1), p10(t_3))$ | SA_ax_30(1.9.3,1.9.4) |
| | infer $post_launch(p9(t_1), p10(t_2))$ | ∨-E(1.6,1.7,1.8,1.9) |
| 2 | $SA_CM(t_2) = launch \wedge \Theta(rd_p9, i, t_1) \wedge \Theta(we_p10, i, t_2) \Rightarrow post_launch(p9(t_1), p10(t_2))$ | ⇒-I |
| | infer $\forall i : Occ, t_1, t_2 : Time \cdot SA_CM(t_2) = launch \wedge \Theta(rd_p9, i, t_1) \wedge \Theta(we_p10, i, t_2) \Rightarrow post_launch(p9(t_1), p10(t_2))$ | ∀∀-I |

■

PROOF (FT_SAFETY_THM2) We then prove the safety theorem holds in the presence of the hypothesised faults.

from $Assumptions, ActivityTheorems, LinkAxioms, FH_2$

1 from $i: Occ, t_1, t_2: Time, \Theta(rd_pX, i, t_1), \Theta(we_pY, i, t_2), FH_2$

1.1 $\Theta(v1_we_pY, i, t_2) \Leftrightarrow \exists j: Occ \cdot \Theta(\Downarrow v1_stateV, j, t_2)$ — v1_ax_1(1.h3,1.h4)

1.2 $\exists j: Occ \cdot \Theta(\Downarrow v1_stateV, j, t_2)$ — $\Leftrightarrow$ -right-E(1.1,1.h3,1.h4)

1.3 $\exists t_3: Time \cdot \Theta(\Uparrow v1_stateV, j, t_3)$ — v1_ax_2(1.2)

infer $post_(pX(t_1), pY(t_2))$ — v1_ax_3(1.2,1.3)

2 $\Theta(rd_pX, i, t_1) \wedge \Theta(we_pY, i, t_2) \wedge FH_2 \Rightarrow post_(pX(t_1), pY(t_2))$ — $\Rightarrow$-I(1)

infer $\forall i: Occ, t_1, t_2: Time \cdot \Theta(rd_pX, i, t_1) \wedge \Theta(we_pY, i, t_2) \wedge FH_2 \Rightarrow post_(pX(t_1), pY(t_2))$ — $\forall$-I(2)

■

PROOF (SAFETY_THM3) We first prove the safety theorem holds under the assumption no faults can occur.

from $Assumptions, ActivityTheorems, LinkAxioms, NOFAULTS$

1 from

1.1 $\Theta(\Uparrow IH_Abandon_Check, j, t_1)$ — IH_rds_p3_ax(1.h3)

1.2 $\Theta(\Downarrow IH_Abandon_Check, j, t_2) \wedge t_1 + u \leq t_2$ — IH_ax_9(1.1)

1.3 $current_mode \neq \text{Abandon} \wedge \Theta(\Uparrow IH_A, k, t_2) \vee$
$current_mode = \text{Abandon} \wedge \Theta(\Uparrow IH_lock_firing, k, t_2)$

1.4 from $current_mode \neq \text{Abandon} \wedge \Theta(\Uparrow IH_A, k, t_2)$

1.4.1 $current_mode \neq \text{Abandon}$ — $\wedge$-E-right(1.4.h1)

infer $\Theta(we_p4, i, t_2)$ — contradiction

1.5 from $current_mode = \text{Abandon} \wedge \Theta(\Uparrow IH_lock_firing, k, t_2)$

1.5.1 $\Theta(\Uparrow IH_lock_firing, k, t_2)$ — $\wedge$-E-left

1.5.2 $\Theta(\Downarrow IH_lock_firing, k, t_3) \wedge t_3 \wedge t_2$ — IH_ax9(1.5.1)

1.5.3 $\Theta(we_p4, i, t_3) \Leftrightarrow \Theta(\Downarrow IH_lock_firing, k, t_3)$ — IH_we_p4_ax

infer $\Theta(we_p4, i, t_2)$ — $\Leftrightarrow$ -E-right(1.5.3,1.5.2)

1.6 $\Theta(we_p4, i, t_2)$ — $\vee$-E(1.3,1.4,1.5)

infer $\exists t_2: Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ — $\exists$-I(1.1.h1,1.6)

2 $Theta(rd_p3, i, t_1) \Rightarrow \exists t_2: Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ — $\Rightarrow$-I(1)

infer $\forall i: Occ, t_1: Time \cdot \Theta(rd_p3, i, t_1) \Rightarrow \exists t_2: Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ — $\forall$-I(2)

■

PROOF (FT_SAFETY_THM3) We then prove the safety theorem holds in the presence of the hypothesised faults.

from $Assumptions, ActivityTheorems, LinkAxioms, FH_3$

| | | |
|---|---|---|
| 1 | from $i : Occ, t_1 : Time, \Theta(rd_p3, i, t_1), FH_1$ | |
| 1.1 | $\Theta(stim_clk4, i, t_2) \wedge \Theta(\Downarrow IH_A, i, t_2) \vee$ <br> $t_1 + l \leq t_2 \leq t_1 + u \wedge \Theta(\Downarrow IH_A, i, t_2)$ | IH_ax_11' |
| 1.2 | from $\Theta(stim_clk4, i, t_2) \wedge \Theta(\Downarrow IH_A, i, t_2)$ <br> as before <br> infer $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | |
| 1.3 | from $t_1 + l \leq t_2 \leq t_1 + u \wedge \Theta(\Downarrow IH_A, i, t_2)$ | |
| 1.3.1 | $\Theta(\Uparrow IH_watchdog, j, t_2)$ | IH_ax_2'(1.3.h1) |
| 1.3.2 | $\Theta(\Downarrow IH_watchdog, j, t_3) \wedge t_3 > t_2)$ | IH_ax_18(1.3.1) |
| 1.3.3 | $\Theta(we_pB, j, t_3)$ | link axiom |
| 1.3.4 | $\Theta(\Uparrow IH_wait, j, t_3)$ | IH_ax_19(1.3.2) |
| 1.3.5 | $\Theta(stim_clk4, k, t_4) \wedge \Theta(\Downarrow IH_wait, j, t_4) \vee$ <br> $\Theta(stim_pB, k, t_4) \wedge \Theta(\Downarrow IH_wait, j, t_4)$ | IH_ax_20 |
| 1.3.6 | from $\Theta(stim_clk4, k, t_4) \wedge \Theta(\Downarrow IH_wait, j, t_4)$ <br> as before <br> infer $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | |
| 1.3.7 | from $\Theta(stim_pB, k, t_4) \wedge \Theta(\Downarrow IH_wait, j, t_4)$ <br> as before <br> infer $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | |
| 1.3.8 | $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | $\vee$-E(1.3.5,1.3.6,1.3.7) |
| | infer $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | |
| 1.4 | $\Theta(we_p4, i, t_2) \wedge t_1 + X \leq t_2$ | $\vee$-E(1.1,1.2,1.3) |
| | infer $\exists t_2 : Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ | $\exists$-I(1.4) |
| 2 | $\Theta(rd_p3, i, t_1) \wedge FH_1 \Rightarrow \exists t_2 : Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ | $\Rightarrow$-I(1) |

infer $\forall i : Occ, t_1 : Time \cdot \Theta(rd_p3, i, t_1) \wedge FH_1 \Rightarrow \exists t_2 : Time \cdot t_1 + X \leq t_2 \wedge \Theta(we_p4, i, t_2)$ — $\forall$-I(2)

■

PROOF (SAFETY_THM5) We first prove the safety theorem holds under the assumption no faults can occur.

from $Assumptions, ActivityTheorems, LinkAxioms, NOFAULTS$

| | | |
|---|---|---|
| 1 | from $i : Occ, t_1 : Time, \Theta(rd_p5, i, t_1)$ | |
| 1.1 | $\Theta(\uparrow IN_A, j, t_1)$ | INav_rs_p5_ax(1.h3) |
| 1.2 | $\Theta(\downarrow write_spec, k, t_3)$ | |
| 1.3 | $\Theta(we_p6, k, t_3) \Leftrightarrow \Theta(\downarrow write_spec, k, t_3)$ | INav_we_p6_ax |
| 1.4 | $\Theta(we_p6, k, t_3)$ | $\Leftrightarrow$ -E-right(1.3,1.2) |
| 1.5 | $\Theta(rd_p9, k, t_4)$ | link axiom |
| 1.6 | $\Theta(\downarrow SA_A, m, t_4)$ | SA_rs_p9_ax(1.5) |
| 1.7 | $\Theta(\downarrow SA_A, m, t_4)$ | |
| 1.8 | $\Theta(\uparrow SA_Check_MS, n, t_4)$ | |
| 1.9 | $\Theta(\downarrow SA_Check_MS, n, t_5)$ | |
| 1.10 | $\Theta(\uparrow SA_Check_Mode, n, t_5)$ | |
| 1.11 | $\Theta(we_p10, n, t_6) \wedge t_6 > t_5$ | |
| | infer $\exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(we_p10, i, t_2)$ | $\exists$-I(1.11) |
| 2 | $\Theta(rd_p5, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(we_p10, i, t_2)$ | $\Rightarrow$-I(1) |
| | infer $\forall i : Occ, t_1 : Time \cdot \Theta(rd_p5, i, t_1) \Rightarrow \exists t_2 : Time \cdot t_2 \leq t_1 + X \wedge \Theta(we_p10, i, t_2)$ | $\forall$-I(2) |

∎

## 8.9 Evaluation

The case study has illustrated that the transformational methodology in earlier chapters is applicable for transforming real-time systems designs, specifically the design method advocated in this thesis: Real Time Networks (RTNs). The fault-tolerant templates described in Chapter 7 have been instantiated in the RTN-SL design to tolerate those faults found by the SHARD analysis. The encapsulated reasoning aided the formal verification task.

Although each template described in Chapter 7 was found to be applicable in the case study, the fine detail required to define each graph grammar transformation made the templates seem over complicated. However, it is accepted that the explicit nature of the RTN-SL specification contributed to this and must therefore be accommodated.

A further criticism is that the formal verification effort is time consuming and should be ideal for mechanical assistance. Given the pre-defined reasoning provided for each template, these conjectures should be readily available with an automated prover for an engineers *tool-kit*.

# Part IV

# Evaluation

# Chapter 9

# Conclusions & Further Work

## Contents


9.1 **Summary** . . . . . 207
9.2 **Evaluation of RTN-SL for specifying faults** . . . . . 208
9.2.1 RTN-SL Language Extensions . . . . . 208
9.2.2 Fault Semantics, $\Omega_f$ . . . . . 209
9.3 **RTN Operational Semantics** . . . . . 209
9.3.1 SOS & LTS . . . . . 210
9.3.2 Conservative Extension . . . . . 210
9.3.3 Soundness . . . . . 211
9.4 **Suitability of graph grammars and application of methodology** . . . . . 212
9.4.1 Transformational methodology . . . . . 212
9.4.2 Case Study Evaluation . . . . . 212
9.5 **Further Work** . . . . . 213
9.6 **Epilogue** . . . . . 215


This chapter summarises the thesis, highlights its contribution and suggests some directions for further work.

## 9.1 Summary

The work reported in this thesis has demonstrated that it is possible to introduce fault tolerance into Real Time Networks in a sound, formally-based manner. In particular, the transformational methodology which has been demonstrated allows practitioners to hypothesise component faults and use well known strategies to provide fault tolerance. Furthermore, these transformations suggest the formal structure with which to reason about emergent fault tolerant properties.

The argument underpinning the thesis was outlined in Chapter 1:

> We argue that transformations, expressed over a graph grammar syntax, do integrate hypothetical fault analysis and design techniques. Further, providing a semantic framework allows to reason about such emergent properties, such as fault tolerance. Such transformations are constrained to respect the well-formedness of designs and their design language principles.
>
> We also argue that extending the existing semantics to make faults explicit, does not violate existing properties of a design.

This thesis has defined and demonstrated an approach to providing a transformational design methodology for RTNs. The RTN architecture makes explicit the distinct components and their interactions with other components through well-defined interfaces within an RTN. This enabled the clear specification of faults to be investigated for each component and provide for fault containment at the component level. Building upon an existing architecture allowed for the focus to be the consideration of faults. Not just the specification of a fault, but the effect this causes on the existing semantics and highlights the changes necessary. This made clear the assumed absence of faults in the existing semantics.

The operational semantics specified for RTNs allowed for the understanding of an RTN in a constructive way. It enabled the step-wise construction of a semantic model which permitted investigations of concurrency and non-determinism. The eventual, two-tier, model separated out the behaviours of *RTN actions* and time. This allowed for the LTS specification-like presentation of the semantics and expressed neatly the behaviours of this concurrent architecture. This was justified by showing the soundness of the existing axiomatic semantics.

The semantic framework for faults and RTNs are shown to be useful in a transformational design methodology expressed in graph grammars. The graph grammar specification of fault tolerant transformations based on the abstract syntax of RTN-SL –itself expressed in a graph grammar– allows for the controlled application of fault tolerance.

The contribution of this thesis lies in four areas:

- A semantic framework to define, tolerate and reason about faults in RTNs, which includes a classification of faults;
- A transformational method for applying well-known fault tolerance strategies to RTN-SL designs;
- An operational model for RTN-SL;
- A soundness argument for the existing axiomatic semantics of RTN-SL with respect to our operational model.

## 9.2 Evaluation of RTN-SL for specifying faults

The applicability of the RTN-SL specification language to support an investigation of faults and support specification of classical fault tolerant strategies is discussed in this subsection. We first consider the RTN-SL language extensions proposed, then the extended axiomatic semantics of the new language features.

### 9.2.1 RTN-SL Language Extensions

Supporting fault tolerance in real-time networks has necessitated extensions to the RTN specification language. Without these extensions the definitions of faults in RTNs would not have been possible. The original language semantics stated only normative behaviours could occur without allowing for the possibility of faults. The extensions we have proposed state explicitly the behaviours we have incorporated. Allowing for faulty behaviours has exposed the assumption that only non-faulty behaviours were considered, regardless of the implementation found. Using Off-the-Shelf (OTS) components may mean this *perfect* implementation is not possible, and the system design not satisfiable. The implications of such an assumption are that it may lead to poverty in some design languages where a set of behaviours feasible are not considered in the design.

We have presented a thorough examination of speculative read and write access methods to IDAs which were introduced into version 3.1 of the RTN-SL language, but not fully defined. This examination incorporated our intention to model faults –internal to activities– as state-machine transitions. That is, *transition guards* specify the conditions in which state-machine transitions can be taken when non-normal behaviour occurs. Specifically, either incomplete read and write accesses at IDAs or faults specify *fault* transitions should be taken, rather than those transitions specified for normal behaviours, or state values, having occurred or being true. By distinguishing fault transitions as reactive measures to faults, we allow ourselves to specify tolerant, containment or corrective actions that should not occur in the absence of faulty behaviours.

### 9.2.2 Fault Semantics, $\Omega_f$

The extended axiomatic semantics that specifies the behaviour of faults in RTNs has been specified that subsume the existing semantics. The existing semantics specified nothing of faults, and crucially, nothing about the omission of faults. It was found that each fault behaviour considered is seen as an alternative behaviour, which is reflected in the structure of the specified extended semantics. Where no faults are considered or feasible, we state this to keep the semantics consistent and sound and provide for a formal framework with which to reason about an RTN.

## 9.3 RTN Operational Semantics

Though an existing axiomatic semantics existed for RTN-SL version 3.1, it was "unclear how to define an operational [or denotational] semantic for RTNs" [Pay02]. This meant the axiomatic semantics remained to be shown sound.

### 9.3.1 SOS & LTS

Our choice to specify an operational semantic for RTNs in a Plotkin SOS style was made for three reasons:

- The set of SOS rules for RTNs form a labelled transition system (LTS) for which we specified a trace semantic model. The trace model allows for conjectures –namely the axiom interpretation– to

be proven inductively over the length of a trace. Although the axiomatic semantics assert the causal behaviours, such as the requirement a leave state event ($\downarrow s$) will follow a state entry event ($\uparrow s$) timely, the associated trace-based inductive assertion must be valid for all traces. That is, should a trace not have reached the length for which the event should have occurred in, then the conjecture would be false. Instead, we only allow for causal assertions if sufficient time has elapsed, i.e. the trace is sufficiently long to include the event. We then reason inductively over the length of the trace to show the soundness of the axiomatic assertions.

- The intuitive appeal of SOS rules allowed for the close examination of each aspect of an RTN (component) in isolation. Each rule specified a single behaviour of an RTN (e.g. read access or state-machine transitions) by stating the condition in which the rule is valid, or *fireable*, and the conclusions state the effect on the RTN state and the events raised. The suitability of the SOS approach is further vindicated by the ease and effectiveness of adding new rules for the fault specifications.
- The LTS approach allowed for the "meta-level", which defines how, and when, each rule can fire to be defined and specify the model of concurrency and non-determinism in RTNs cleanly.

### 9.3.2 Conservative Extension

It has been shown that additional language features proposed do not conflict with the existing semantics through the result that the extended TTS is a conservative extension to the existing semantics. This result was argued with the understanding that faults are alternative behaviours and, given the hypothesis that no faults occur, then existing axioms in the existing semantics remain true. The conservativity result supports our approach to defining the semantics in an incremental way and justifies our decision to use an SOS style approach.

### 9.3.3 Soundness

The claim that something has been proven should eliminate doubt. Unfortunately, informal arguments cannot create such certainty. Our ambition to show the axiomatic semantics sound with respect the operational model therefore requires an argument, or proof, which convinces the reader of its correctness. Two options exist: i) A *Rigorous Proof* which borrows the ideas of carefully structured and line-by-line justifications from a *formal proof*, but omits obvious hypotheses, uses abbreviations or appeals to general theories for justifications rather than to specific rules of inference; ii) A *Mechanised Proof* is one which each step and inference rule is encoded in the logical framework of a prover and each step is justified by appealing to a formally stated rule of inference.

We elected to present rigorous proofs as evidence the axiomatic semantics are sound, and are confident these proofs can be made formal or verified (as described below) by a proof tool. The benefits of this approach enable the proofs to be readable, such that a reader can follow and understand the intuition, rather than having to accept the correctness of the tool – if the tool is incorrect, or the encoding of our semantics and validation conjectures misrepresented, then the verification by the tool is worthless.

Showing the axiomatic semantics sound with respect our operational model has reiterated the correctness which has previously been inferred from the axioms being usable in a formal proof. However, our attempts

to show the soundness provides a firm rigorous argument irrespective of past experience and examples and allows for the examination of our argument, rather than searching for a counter example. However, as outlined in Section 6.5, a formalised proof of our soundness arguments, possibly using Isabelle/HOL or PVS, would support our claims of soundness.

Having argued the axiomatic semantics are sound, we can now proceed with verification of RTN-SL designs as described previously using the PVS Theorem Prover.

## 9.4 Suitability of graph grammars and application of methodology

We have proposed a transformational design methodology, using a context-sensitive graph grammar, which permits the removal of a faulty design component from its host design and its replacement by an arrangement of non-ideal components which provide for a tolerance to some hypothetical fault. We examine the suitability of using a graph grammar and the application of this methodology (using the case-study) below.

### 9.4.1 Transformational methodology

Providing for graph grammar productions as our transformational method, we allow for a step-wise approach. That is, a design is transformed to the next by applying one (or more) productions that replace one (or more) components with an arrangement of design components. This allows a designer to explore, for a given context, each valid production to decide the best trade-off between the cost of a fault tolerance strategy and the improved reliability sought.

Although the design templates we propose have been successfully applied to a realistic design, it is anticipated the flat graph grammar will not be sufficient for more complex templates. Rather than the graph grammar not being expressive enough to specify such templates, it is thought the designs produced will be intractable for cohesion with the existing design and the designs will not suggest a structure for the validation which is beneficial.

### 9.4.2 Case Study Evaluation

To test the applicability of our design methodology and suitability of representing a RTN in a graph grammar syntax, we undertook a sizeable case-study. The results of this study are summarised below:

- The graph grammar syntax was suitable for representing the graphical syntax of RTN-SL, but not the supporting concrete syntax;
- The necessity of a context-sensitive grammar allowed for the derivation of standard RTN designs;
- It was realised, as hoped, that the formal reasoning was structured as the productions were presented, often properties prescribed of a templates were used as lemma's when reasoning at the activity and network level;
- Although the structure of each template provided insight into the proof effort, the repetition was tiresome and it is expected the repetitive steps are appropriate for automation.

- Acquiring the graph grammar representation of a RTN-SL design should be automated.

Overall, the case study has justified our decisions to use RTN-SL, graph grammars and the PVS theorem prover to show the applicability of this methodology.

## 9.5 Further Work

**Hierarchical Graph Grammars:** Our attempts to represent a structural RTN in a *flat* graph grammar has caused the loss of the RTN structure. Instead of an activities state-machine being seen as internal to an activity at the RTN level, our grammar shows the flattened RTN, with all components visible. This makes the RTN level transformations more tiresome –yet possible– whereas it is considered a hierarchical grammar would retain the RTN structure and make the RTN level transformations more tractable.

**Modal Logic:** Our choice to use RTL to model faults was primarily decided to give compatibility with the RTN-SL semantics, which are specified in RTL. Another consideration may be to use a modal logic, as it seems attractive to consider operators such as *eventually*. Given our observation that faults are visible for distinct intervals, then these operators would seem appropriate, for example, to specify a late fault is to say a value is *eventually* produced after some specific time, otherwise it would be an omission fault.

Paynter, in [Pay03], has begun to develop a new modal logic called Causal Real Time Logic (CRTL) which adapts the existing RTL logic to be capable of specifying and reasoning about causality in real-time networks (RTNs). The approach to model the *history* of an RTN is similar to our concept of a *trace*.

**Isabelle/HOL:** To mechanically verify our soundness proofs, as suggested in Section 6.5, would add further conviction to our argument. Although the rigorous arguments we have give are believed to be true, it would be an interesting exercise to take the next obvious step and use a theorem prover, such as Isabelle/HOL, to verify our conjectures. It would also be worth considering undertaking this task using the PVS theorem prover, which may then lead to an integration with the existing axiomatic semantics.

Using such a mechanised approach may also provide the opportunity to show the axioms *complete*, but such an approach is not known to-date.

**Further fault tolerant templates:** Those templates proposed in this thesis are purposefully simple and only of well-known strategies to allow for the development of the methodology. It would now be interesting to examine other, more novel, strategies to test the bounds of the methodology. Devising new strategies specific to RTNs would be feasible given the structure of our approach to document and test new ideas.

**Theorem/Proof Generator:** As reported elsewhere [OE02], an integrated environment which interacts with the RTN-SL tool support, NetSpec and provides some graph grammar support for generating graph grammar representations of a design to allow a designer to apply our fault tolerant templates would provide the foundation for a larger objective: a theorem and proof generator. Given the structured approach of our methodology and our experience with the case-study, applying a transformation to a RTN design, it

should be possible to i) generate the theorems (or lemmas) that show a template is functionally equivalent to the existing components, and ii) discharge these lemmas with pre-defined tactics. Providing this type of environment would allow for the seamless introduction of our, and other, design templates.

## 9.6 Epilogue

Our initial objective was to investigate the addition of fault tolerance into Real-Time Networks. This necessitated the study and definition of faults in a real-time system architecture and the survey of existing fault tolerant strategies applicable to RTS.

The methodologies developed have found the fault definitions, strategies and design methods applicable to designing fault tolerant RTNs. We have extended the semantic framework for RTNs and used it to first show the axiomatic semantics sound, then to use the axiomatic semantics to verify the transformed design are tolerant to the faults considered.

# Bibliography


[ABV94] Luca Aceto, Bard Bloom, and Frits Vaandrager, *Turning sos rules into equations*, Inf. Comput. **111** (1994), no. 1, 1–52.

[AFV99] L Aceto, W Fokkink, and C Verhoef, *Structural Operational Semantics*, Elsevier, March 1999.

[AK98a] A Arora and S S Kulkarni, *Component based design of multitolerant systems*, Software Engineering **24** (1998), no. 1, 63–78.

[AK98b] ———, *Detectors and correctors: A theory of fault-tolerance components*, International Conference on Distributed Computing Systems, 1998, pp. 436–443.

[AKTY99] Y Adachi, S Kobayashi, K Tscuchida, and T Yaku, *An NCE context-sensitive graph grammar for visual design languages*, IEEE Symposium on Visual Languages, 1999.

[Arm98] J M Armstrong, *Industrial integration of graphical formal specifications*, Journal of Systems Software **20** (1998), 211–225.

[Bar96] J Barnes, *Programming in Ada-95*, Addison-Wesley, 1996.

[BFG99] C Bernardeschi, A Fantechi, and S Gnesi, *Formal Validation of the GUARDS Inter-Consistency Mechanism* , Proc. 18th Int. Conf. on Computer Safety, Realiability and Security, Lect. Notes Comput. Sci., vol. 1698, Springer-Verlag, 1999, pp. 420–430.

[BFL+94] J C Bicarregui, J S Fitzgerald, P A Lindsay, R Moore, and B Ritchie, *Proof in VDM: A Practitioner's Guide*, Formal Approaches to Computing and Information Technology (FACIT), Springer-Verlag, 1994.

[Bor98] Robert W Born, *Patterns for safety-critical systems*, Master's thesis, University of York, 1998.

[Bra03] Ian Bray, *An introduction to requirements engineering*, Addison-Wesley, 2003.

[BS90] A Bondavalli and L Simoncini, *Failure classification with respect to detection*, First Year Report, Task B: Specification and Design for Dependability, ESPRIT BRA PRoeject 3092: Predictably Dependable Computing Systems, vol. 2, May 1990.

[BSS94] A Bondavalli, J Stankovic, and L Strigini, *Adaptable fault tolerance for real-time systems.*, Tech. Report UM-CS-1994-039, University of Massachusetts, Amherst, May 1994.

[BSW69] K A Barlett, R A Scantlebury, and P T Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Communications of the ACM **12** (1969), no. 5.

[CIS77] CISHEC, *A guide to hazard and operability studies (HAZOP)*, The Chemical Industry Safety and HEalth Council of the Chemical Industries Associations Ltd., 1977.

[CM92] J Camilleri and T Melham, *Reasoning with inductively defined relations in the HOL theorem prover*, Tech. Report 265, Computer Laboratory, University of Cambridge, August 1992.

[Cri85] Flaviu Cristian, *A rigorous approach to fault-tolerant programming*, IEEE Transactions on Software Enginerring **11** (1985), no. 1.

[Cri91] ———, *Understanding fault-tolerant distributed systems*, Communications of the ACM **34** (1991), no. 2, 56–78.

[Cri94] ———, *Abstractions for fault-tolerance*, Proceedings of the IFIP 13th World Computer Congress. Volume 3 : Linkage and Developing Countries (Amsterdam, The Netherlands) (Karen Duncan and Karl Krueger, eds.), Elsevier Science Publishers, 1994, pp. 278–286.

[dH01] L de Alfaro and T A Henzinger, *Interface theories for component-based design*, Lecture Notes in Computer Science **2211** (2001), 148.

[Eng95] U Engberg, *Reasoning in the Temporal Logic of Actions - The design and implementation of an interactive computer system*, Ph.D. thesis, Department of Computer Science, University of Aarhus, September 1995.

[ES85] P D Ezhilchelvan and S K Shrivastava, *A characterisation of faults in systems*, Tech. Report 206, University of Newcastle, September 1985.

[FL98] John Fitzgerald and Peter Gorm Larsen, *Modelling systems: practical tools and techniques in software development*, Cambridge University Press, 1998.

[Fuc92] N E Fuchs, *Specifications are (preferably) executable*, IEE/BCS Software Engineering Journal, vol. 7, 323–334, 1992.

[FV98] Wan Fokkink and Chris Verhoef, *A conservative look at operational semantics with variable binding*, Information and Computation **146** (1998), no. 1, 24–54.

[Har97] P Hartel, *LATOS – a lightweight animation tool for operational semantics*, Tech. Report DSSE-TR-97-1, University of Southampton, 1997.

[Hay93] I. J. Hayes (ed.), *Specification case studies*, 2nd ed., Prentice Hall International Series in Computer Science, 1993.

[HJ89] I J Hayes and C B Jones, *Specifications are not (necessarily) executable*, Software Engineering Journal **4** (1989), no. 6, 320–338.

[HN96] D Harel and A Naamad, *The STATEMATE semantics of statecharts*, ACM Transactions on Software Engineering and Methodology **5** (1996), no. 4, 293–333.

[Hoo97] J Hooman, *Compositional verification of real-time applications*, Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97) (Bad Malente, Germany) (Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, eds.), vol. 1536, Springer-Verlag, 1997, pp. 276–300.

[IoMJ87] Joint IECCA and MUF Committee on MASCOT (JIMCOM), *The official Handbook of MASCOT: Version 3.1 - Issue 1*, June 1987.

[JM86] F Jananian and A K Mok, *Safety analysis of timing properties in real-time systems*, IEEE Transactions on Software Engineering **12** (1986), no. 9.

[JMS88] F Jananian, A K Mok, and D A Stuart, *Formal specification of real-time systems*, Tech. Report TR-88-25, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, June 1988.

[Jon03] C B Jones, *Operational Semantics: Concepts and their Expression*, Information Processing Letters **88** (2003), 27–32.

[Jon04] ______, *Csc334 course notes*, 2004.

[KA00] S S Kulkarni and A Arora, *Automating the addition of fault-tolerance*, Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings (Pune, India) (Mathai Joseph, ed.), no. 1926, Springer-Verlag, 2000, pp. 82–93.

[KRS99] S S Kulkarni, J Rushby, and N Shankar, *A case-study in component-based mechanical verification of fault-tolerant programs*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA (1999), 33–40.

[LA90] Pete A Lee and Tom Anderson, *Fault tolerance: Principles and practice*, Springer-Verlag, 1990.

[Lam94] L Lamport, *The temporal logic of actions*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 3, 872–923.

[Lap92] J C Laprie (ed.), *Dependability: Basic Concepts and Terminology*, Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer-Verlag, 1992.

[LHP+96] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, Parkin Parkin, et al., *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language*, December 1996.

[LJ92] Z Liu and M Joseph, *Transformations of programs for fault-tolerance*, Formal Aspects of Computing **4** (1992), no. 5, 442–469.

[LJ96] ______, *Verification of fault tolerance and real time*, Proceedings of the 26th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26) (Sendai, Japan), IEEE, 1996, pp. 220–229.

[LJ97] ———, *Formalizing real-time scheduling as program refinement*, ARTS, 1997, pp. 295–309.

[LJ99] ———, *Specification and verification of fault-tolerance, timing, and scheduling*, ACM Trans. Program. Lang. Syst. **21** (1999), no. 1, 46–89.

[LM94] L Lamport and S Merz, *Specifying and verifying fault-tolerant systems*, FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems – ProCoS, LNCS, Springer-Verlag, 1994.

[Mil80] Robin Milner, A Calculus of Communicating Systems, vol. 92, Springer, 1980.

[Min96] Ministry of Defence, *Safety management requirements for defence systems*, Defence Standard 00-56(Part 1) **Issue 2** (1996).

[Min97] ———, *Requirements for safety related software in defence equipment*, Defence Standard 00-55(Part 1) **Issue 2** (1997).

[Min99] ———, *Requirements for safety related electronic hardware in defence equipment*, Defence Standard 00-54(Part 1) **Issue 1** (1999).

[MNPF95] John A McDermid, M Nicholson, David J Pumfrey, and P Fenelon, *Experience with the application of HAZOP to computer-based systems*, COMPASS '95: Proceedings of the Tenth Annual Conference on Computer Assurance, IEEE, 1995, pp. 37–48.

[MP94] John A McDermid and David J Pumfrey, *A development of hazard analysis to aid software design*, Compass'94: 9th Annual Conference on Computer Assurance (Gaithersburg, MD), National Institute of Standards and Technology, 1994, pp. 17–26.

[MP98] ———, *Safety analysis of hardware / software interactions in complex systems*, Proceedings of the 16th International System Safety Conference, 1998, pp. 232–241.

[Nip98] Tobias Nipkow, *Winskel is (almost) right: Towards a mechanized semantics textbook*, Formal Aspects of Computing **10** (1998), 171–186.

[Nor92] J Nordahl, *Design for dependability*, Dependable Computing for Critical Applications 3 (C E Landwehr, B Randell, and L Simoncini, eds.), Dependable Computing and Fault-Tolerant Systems, vol. 8, Springer-Verlang, 1992, pp. 65–89.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel, *Isabelle/hol — a proof assistant for higher-order logic*, LNCS, vol. 2283, Springer, 2002.

[OE02] Dan J Owen and Paul D Ezhilchelvan, *Verifiable fault-tolerant transformation of a real-time legacy system*, Tech. Report CS-TR-785, University of Newcastle, 2002.

[OSRSC99a] S Owre, N Shanker, J Rushby, and D W J Stringer-Calvert, *PVS Language: Version 2.3*, Tech. report, Computer Science Laboratory - SRI International, September 1999.

[OSRSC99b] ———, *PVS System Guide: Version 2.3*, Tech. report, Computer Science Laboratory - SRI International, September 1999.

[PABD+99] D Powell, J Arlat, L Beus-Dukic, A Bondavalli, P Coppola, A Fantechi, E Jenn, C Rabejac, and A Wellings, *GUARDS: a generic upgradable architecture for real-time dependable systems*, Parallel and Distributed Systems, IEEE Transactions on **10** (1999), no. 6, 580–599.

[PAH00] Stephen E Paynter, Jim M Armstrong, and Jan Haveman, *ADL: An Activity Description Language for Real-Time Networks*, Aspects of Computing **12** (2000), no. 2.

[Pay93] Stephen E Paynter, *The Formalisation of Software Development Using MASCOT*, Ph.D. thesis, Mathematics Department, University of Southampton, September 1993.

[Pay95] ______, *Structuring the semantic definitions of graphical design notations.*, IEE Software Engineering Journal (1995).

[Pay01a] S.E. Paynter, *A BVRAAM Case Study in Safety Engineering: Formal Methods*, Tech. Report DR 23123, Matra BAe Dynamics, December 2001.

[Pay01b] ______, *A BVRAAM Case Study in Safety Engineering: Safety Analysis*, Tech. Report DR 23122, Matra BAe Dynamics, December 2001.

[Pay01c] ______, *A BVRAAM Case Study in Safety Engineering: Specification and Design*, Tech. Report DR 23121, Matra BAe Dynamics, December 2001.

[Pay01d] Stephen E Paynter, *NetSpec tool*, November 2001.

[Pay02] ______, *RTN-SL: The Real-Time Network Specification Language. Version 3.1*, Tech. Report DR 20656, MBDA, March 2002.

[Pay03] ______, *Towards causal real-time logic: Adding counterfactuals, variables, and locality to events in RTL*, Tech. Report DR 26304, MBDA, May 2003.

[Plo81] Gordon Plotkin, *A structural approach to operational semantics*, Tech. Report DAIMI FN-19, Department of Computer Science, University of Aarhus, Denmark, 1981.

[Plo03] ______, *The Origins of Structural Operational Semantics*, 2003.

[Pra84] K V S Prasad, *Specification and proof of a simple fault tolerant system in CCS*, Tech. Report Internal Report CSR–178–84, University of Edinburgh, December 1984.

[Roz87] G Rozenberg, *An introduction to the NLC way of rewriting graphs*, Proceedings of Graph Grammars and Their Application to Computer Science (H Ehrig, M Nagl, G Rozenberg, and A Rosenfeld, eds.), Lecture Notes in Computer Science, vol. 291, 1987, pp. 55–66.

[SH94] H Schepers and J Hooman, *A trace-based compositional proof theory for fault tolerant distributed systems*, Theoretical Computer Science **128** (1994), no. 1–2, 127–157.

[SHS+97] Christian Scheidler, Günter Heiner, Ralph Sasse, Emmerich Fuchs, Hermann Kopetz, and Christopher Temple, *Time-Triggered Architecture (TTA)*, Advances in Information Technologies: The Business Challenge (1997).

[Sim86] Hugo R Simpson, *The MASCOT Method*, Software Engineering Journal, 1986 1986.

[Sim94] ———, *Methodological and Notational Conventions in DORIS Real-Time Networks*, IED Supporting Predictable Implementation of Requirements in Timing and Safety (SPRINTS) Deliverable, 1994.

[Sim03] ———, *Protocols for process interaction*, IEE Proc.-Comput. Digit. Tech. **150** (2003), no. 3.

[SS83] R D Schlichting and F B Schneider, *Fail-stop processes: An approach to designing fault-tolerant computing systems*, ACM Transactions on Computer Systems **1** (1983), no. 3, 222–238.

[tea94] Centaur team, *Centaur tutorial*, Tech. report, INRIA Rocquencourt, France, July 1994.

[vDHK96] A van Deursen, J Heering, and P Klint, *Language Prototyping: An Algebraic Specification Approach*, AMAST Series in Computing, vol. 5, World Scientific, Singapore, 1996.

[WD96] J. C. P. Woodcock and J. Davies, *Using Z: Specification, proof and refinement*, Prentice Hall International Series in Computer Science, 1996.

[Win93] Glynn Winskel, *The formal semantics of programming languages: an introduction*, MIT Press, 1993.

[WLG+78] J H Wensley, L Lamport, J Goldberg, M W Green, K N Levitt, P M Melliar-Smith, R E Shostak, and C B Weinstock, *SIFT: Design and analysis of a fault-tolerant computer for aircraft control*, IEEE, vol. 66, October 1978, pp. 1240–1255.

[Woo96] G Woodward, *Rapier 2000 software development programme*, Software Engineering Journal **11** (1996), no. 2.

# Index


*ES*-indn, 102
$T_0$, 81
$T_1$, 86
$\Omega_f$, 69
$\Pi$, 78
$\Pi_{rtn}$-indn, 102
$\Sigma$, 78
$\mathcal{M}$, 96
$\xrightarrow{rtn}$, 80
$\xrightarrow{s}$, 80
*step* rule, 80
*tick* rule, 80

Commission Fault, 19
Completeness, 93
Component, 12
Component Fault, 16
Conservative Extension, 89
Context-sensitive graph grammar, 36
Correct Behaviour, 17

Design, 14
Design Fault, 16

Environment, 11
Erroneous State, 15
Erroneous Transition, 15
Error, 15

Fail Stop (Activity), 130
Fail-Signal (Activity), 129
Failure, 13
Fault Hypothesis (FH), 19
Fault Taxonomy, 20

Graph, 37

Hierarchical Graph Grammars, 166

Inductive Definitions, 93
Interface, 11
Isabelle/HOL, 166

Labelled Transition Systems (LTS), 76

MASCOT, 24
Modal Logic, 166

Omission Fault, 18
Operational Conservative Extension, 88

Passive State Replication (PSR), 21
Plotkin-style Transition Rules, 77

Real-Time Logic (RTL), 52
Redundancy, 21
RTN-SL, 24
RTN-Types, 78

SHARD, 29
Soundness, 93
Source-dependency, 89
Specification, 13
System, 11, 12

Temporal Redundant IDA, 128
Timing Fault, 18
Trace Induction, 99
Triple Modular Redundancy (TMR), 22

Value Fault, 19

Watchdog Timer Template, 126

# Part V

# Supplementary Material

# Appendix A

# Case Study Specifications

We present here the complete RTN-SL specification of the case study design. The subsequent subsection details the modified axiomatic semantics generated by the transformations described in Chapter 8.

## A.1 RTN-SL Design

**Thesis–caseStudyHostDesign.rtnsl**

```
---------------------------
--                       --
-- Begin ADT Descriptions --
--                       --
---------------------------

adt Image is
   type Raw_Image is token;
   type INS_Data is token;
   type Processed_Image is token;
   type Vector is (x,y,z);

   function Image_Processing(ri : Raw_Image) return pi : Processed_Image;
   pre true;
   post true;  -- The functionality of this function is not given.
end adt;

adt BIT_Test is
  with Image;
   type BIT_Assessment is (pass, fail);

   function Assessment(ri:Image.Raw_Image) return ass : BIT_Assessment;
   pre true;
   post true;  -- The functionality of this function is not given.
end adt;

adt Threat is
   type Threat_Assessment is (high, medium, low);
end adt;

adt Pos is
  type coords is token;

  function calc_pos(ri:Image.Raw_Image) return ps : coords;
  pre true;
  post true;
end adt;

adt INS_Data is
 with Image;
  type pos is Image.Vector;
  type msg is token;
end adt;

adt Missile_State is
 with Image;
  type pos is Image.Vector;
  type status is (model, mode2);
  type IMU_Data is token;

---------------------------------------------------------------------
--                                                                 --
-- FUNCTION: compare()                                             --
--
-- DESC: ...
--                                                                 --
---------------------------------------------------------------------

  function compare (pi : Image.Processed_Image) return res : bool;
  pre true;
  post true;  -- The functionality of this function is not given.

end adt;

adt Mode is
  type number is nat;
end adt;

adt Reports is
  type status is token;
end adt;
```



**Thesis–caseStudyHostDesign.rtnsl**

```
adt Actuators is
  type cmd is token;
  type pos is token;
end adt;

adt Firing is
  type lock is token;
end adt;

adt Clock is
  type tick is token;
end adt;


---------------------------------
--                             --
-- Begin Activity Descriptions --
--                             --
---------------------------------


---------------------------------------------------------------------------
--                                                                       --
-- MC Specification                                                      --
-- ================                                                      --
-- The "Mode Controller" (MC) activity loops...
---------------------------------------------------------------------------

activity MC is
  with Clock, Mode;

  ports
     p0 : (Stimulus, NULL, out);
     p1 : (Channel, Mode.number, in);
     p2 : (Pool, Mode.number, out);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  local state
     current_mode : Mode.number;
  end local state;

  operations
     op check_mode (new_mode : Mode.number);
      ext write current_mode;
       pre true;
       post true;
     end op;

     op set_timeout () start : NULL;
       pre true;
       post true;
     end op;

     op raise_mode () ev : Mode.number;
      ext read current_mode;
       pre true;
       post true;
     end op;
  end operations;

  states
     static MC_A
        transition goes to MC_Check_Mode on read p1;
     end state;

     dynamic MC_Check_Mode
```

**Thesis–caseStudyHostDesign.rtnsl**

```
         op check_mode reads from p1;
         timing [l,wcet,u];
         transition goes to MC_Set_Timeout on current_mode = "TIMEOUT";
         transition goes to MC_Raise_Mode on current_mode /= current_mode~;
      end state;

      dynamic MC_Set_Timeout
         op set_timeout writes to p0;
         timing [l,wcet,u];
         transition goes to MC_A on true;
      end state;

      dynamic MC_Raise_Mode
         op raise_mode writes to p2;
         timing [l,wcet,u];
         transition goes to MC_A on true;
      end state;

      initial MC_A;
   end states;

end activity;


--------------------------------------------------------------------------
--                                                                      --
-- IH Specification                                                     --
-- ================                                                     --
-- The "Interlock Handler" (IH) activity loops...
--------------------------------------------------------------------------


activity IH is
   with Mode, Firing;

   ports
      p3    : (Pool, Mode.number, in);
      p4    : (Pool, Firing.lock, out);
      pclk4 : (Stimulus, Clock.tick, in);
   end ports;


   auxiliary definitions
      constant l : Time;
      constant wcet : Time;
      constant u : Time;
   end auxiliary definitions;

   local state
      current_mode : Mode.number;
   end local state;

   operations
      op abandon_check (new_mode : Mode.number);
       ext write current_mode;
        pre true;
        post current_mode = new_mode;
      end op;

      op lock_firing () lock : Firing.lock;
       ext read current_mode;
        pre true;
        post true;
      end op;
   end operations;

   states

-- should be a STIM not read

      static IH_A
         transition goes to IH_Abandon_Check on read pclk4;
      end state;
```

**Thesis–caseStudyHostDesign.rtnsl**

```
      dynamic IH_Abandon_Check
         op abandon_check reads from p3;
         timing [l,wcet,u];
         transition goes to IH_A on false; --i.e. NOT abandon
         transition goes to IH_Lock_Firing on true; --i.e. abandon
      end state;

      dynamic IH_Lock_Firing
         op lock_firing writes to p4;
         timing [l,wcet,u];
         transition goes to IH_A on true;
      end state;

      initial IH_A;
   end states;

end activity;

--------------------------------------------------------------------------
--                                                                      --
-- IN Specification                                                     --
-- ================                                                     --
-- The "Inertial Navigation" (IN) activity loops...
--------------------------------------------------------------------------


activity INav is
   with Image;

   ports
      p5 : (Signal, Image.Processed_Image, in);
      p6 : (Signal, Missile_State.status, out);
      p7 : (Channel, Mode.number, out);
      p8 : (Pool, Missile_State.pos, out);
   end ports;

   auxiliary definitions
      constant l : Time;
      constant wcet : Time;
      constant u : Time;
   end auxiliary definitions;

   local state
      body_motion : Image.Processed_Image;
   end local state;


   operations
      op integrity_mode (new_bm : Image.Processed_Image);
       ext write body_motion;
        pre true;
        post true;
      end op;

      op write_seperation () seperation_cmd : Missile_State.status;
       ext read body_motion;
        pre true;
        post true;
      end op;

      op write_alignment () align : Missile_State.pos;
       ext read body_motion;
        pre true;
        post true;
      end op;

      op raise_mode () ev : Mode.number;
       ext read body_motion;
        pre true;
        post true;
      end op;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
  end operations;

  states
     static IN_A
        transition goes to IN_Integrity_Mode on read p5;
     end state;

     dynamic IN_Integrity_Mode
        op integrity_mode reads from p5;
        timing [l,wcet,u];
        transition goes to IN_A on false; --i.e. fail
        transition goes to IN_Write_Seperation on true; --i.e. pass
     end state;

     dynamic IN_Write_Seperation
        op write_seperation writes to p6;
        timing [l,wcet,u];
        transition goes to IN_Write_Alignment on true;
     end state;

     dynamic IN_Write_Alignment
        op write_alignment writes to p8;
        timing [l,wcet,u];
        transition goes to IN_Raise_Mode on true;
     end state;

     dynamic IN_Raise_Mode
        op raise_mode writes to p7;
        timing [l,wcet,u];
        transition goes to IN_we_p7 on true;
     end state;

     static IN_we_p7
        transition goes to IN_A on write p7;
     end state;

     initial IN_A;
  end states;

end activity;

---------------------------------------------------------------------------
--                                                                      --
-- SA Specification                                                     --
-- ================                                                     --
-- The "Seperation Autopilot" (SA) activity loops...
---------------------------------------------------------------------------

activity SA is
  with Image, Actuators, Missile_State, Mode;

  ports
     p9  : (Signal, Missile_State.status, in);
     p10 : (Signal, Actuators.cmd, out);
     p11 : (Pool, Mode.number, in);
     p12 : (Pool, Image.Processed_Image, in);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  local state
     MS : Missile_State.status;
     CM : Mode.number;
     traj : Image.Processed_Image;
  end local state;

  operations
     op check_MS (current_ms : Missile_State.status);
```

**Thesis-caseStudyHostDesign.rtnsl**

```
      ext write MS;
       pre true;
       post MS = current_ms;
     end op;

     op check_mode (current_mode : Mode.number);
      ext write CM;
       pre true;
       post CM = current_mode;
     end op;

     op init_act () init_cmd : Actuators.cmd;
      ext read MS, CM;
       pre true;
       post CM = 4 implies init_cmd = "init";
     end op;

     op check_traj (current_traj : Image.Processed_Image);
      ext write traj;
       pre true;
       post traj = current_traj;
     end op;

     op launch () launch_cmd : Actuators.cmd;
      ext read MS, CM;
       pre true;
       post true;
     end op;
  end operations;

  states
     static SA_A
        transition goes to SA_Check_MS on read p9;
     end state;

     dynamic SA_Check_MS
        op check_MS reads from p9;
        timing [l,wcet,u];
        transition goes to SA_Check_Mode on true;
     end state;

     dynamic SA_Check_Mode
        op check_mode reads from p11;
        timing [l,wcet,u];
        transition goes to SA_A on CM /= "LAUNCH";
        transition goes to SA_Init_Act on CM = "INIT";
        transition goes to SA_Calc_Traj on CM = "LAUNCH";
     end state;

     dynamic SA_Calc_Traj
        op check_traj reads from p12;
        timing [l,wcet,u];
        transition goes to SA_Launch on true;
     end state;

     dynamic SA_Init_Act
        op init_act writes to p10;
        timing [l,wcet,u];
        transition goes to SA_A on true;
     end state;

     dynamic SA_Launch
        op launch writes to p10;
        timing [l,wcet,u];
        transition goes to SA_A on true;
     end state;

     initial SA_A;
  end states;

end activity;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
--------------------------------------------------------------------------
--                                                                      --
-- RAM Specification                                                    --
-- ==================                                                   --
-- Desc:
--------------------------------------------------------------------------

activity RAM is
  with Image, Pos, BIT_Test, INS_Data, Mode;

  ports
     p13  : (Signal, Image.Raw_Image, in);
     p14  : (Pool, Pos.coords, out);
     p15  : (Channel, Mode.number, out);
     p16  : (Pool, Image.Processed_Image, out);
     p17  : (Signal, INS_Data.pos, out);
     p18  : (Pool, Pos.coords, out);
     p19  : (Signal, BIT_Test.BIT_Assessment, out);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  local state
     input : Image.Raw_Image;
  end local state;

  operations
     op Read_Image (raw : Image.Raw_Image);
      ext write input;
        pre true;
        post input = raw;
     end op;

     op Process_Image () processed : Image.Processed_Image;
      ext read input;
        pre true;
        post processed = Image.Image_Processing(input);
     end op;

     op Raise_Event () evid : Mode.number;
      ext read input;
        pre true;
        post evid = 5;
     end op;

     op BIT () verdict : BIT_Test.BIT_Assessment;
      ext read input;
        pre true;
        post verdict = BIT_Test.Assessment(input);
     end op;

     op Record_input () output : Image.Processed_Image;
      ext read input;
        pre true;
        post output = Image.Image_Processing(input);
     end op;

     op Write_Position () position : Pos.coords;
      ext read input;
       pre true;
       post position = Pos.calc_pos(input);
     end op;

     op Write_INS_Data () INS_msg : INS_Data.pos;
      ext read input;
       pre true;
       post true;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
     end op;
  end operations;

  states
     static RAM_A
        transition goes to RAM_B on read p13;
     end state;

     dynamic RAM_B
        op        Read_Image reads from p13;
        timing    [l, wcet, u];
        transition goes to umb_cut on input = "umb_cut";
        transition goes to BIT_cmd on input = "BIT_cmd";
        transition goes to write_input on input /= ("umb_cut" OR "BIT_cmd");
     end state;

     dynamic umb_cut
        op        Raise_Event writes to p15;
        timing    [l, wcet, u];
        transition goes to umb_cut_we on true;
     end state;

     static umb_cut_we
        transition goes to RAM_A on write p15;
     end state;

     dynamic BIT_cmd
        op        BIT writes to p19;
        timing    [l, wcet, u];
        transition goes to RAM_A on true;
     end state;

     dynamic write_input
        op        Write_Position writes to p14;
        op        Process_Image writes to p16;
        op        Write_INS_Data writes to p17;
        op        Write_Position writes to p18;
        timing    [l, wcet, u];
        transition goes to RAM_A on true;
     end state;

     initial RAM_A;
  end states;
end activity;


--------------------------------------------------------------------------
--                                                                      --
-- TA Specification                                                     --
-- ================                                                     --
-- The "Transfer Alignment" (TA) activity loops...
--------------------------------------------------------------------------

activity TA is
  with Image, INS_Data, Missile_State, Reports;

  ports
     p20  : (Signal, INS_Data.pos, in);
     p21  : (Pool, Missile_State.pos, in);
     p22  : (Channel, Mode.number, out);
     p23  : (Signal, Reports.status, out);
     p24  : (Pool, INS_Data.msg, out);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  local state
     input : INS_Data.msg;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
    MS : Missile_State.status;
  end local state;

  operations
    op Read_Image (inp : INS_Data.pos);
     ext write input;
      pre true;
      post input = inp;
    end op;

    op Check_Missile_Alignment (incoming_ms : Missile_State.pos);
     ext write ms;
      pre true;
      post MS = incoming_ms;
    end op;

    op Generate_Report () rp : Reports.status;
     ext read MS;
      pre true;
      post true;
    end op;

    op Raise_Event () ev : Mode.number;
     ext read MS;
      pre true;
      post true;
    end op;

    op Write_INS () output : INS_Data.msg;
     ext read input;
      pre true;
      post true;
    end op;
  end operations;

  states
    static TA_A
      transition goes to TA_B on read p20;
    end state;

    dynamic TA_B
      op Read_Image reads from p20;
      timing [l,wcet,u];
      transition goes to TA_C on true;
    end state;

    dynamic TA_C
      op Check_Missile_Alignment reads from p21;
      timing [l,wcet,u];
      transition goes to TA_D on true;
    end state;

    dynamic TA_D
      op Generate_Report writes to p23;
      timing [l,wcet,u];
      transition goes to TA_E on true;
    end state;

    dynamic TA_E
      op Raise_Event writes to p22;
      timing [l,wcet,u];
      transition goes to TA_F on true;
    end state;

    static TA_F
      transition goes to TA_G on write p22;
    end state;

    dynamic TA_G
      op Write_INS writes to p24;
      timing [l,wcet,u];
      transition goes to TA_A on true;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
    end state;

    initial TA_A;
  end states;
end activity;


---------------------------------------------------------------------------
--                                                                       --
-- MB Specification                                                      --
-- ================                                                      --
-- The "Manage BIT" (MB) activity loops...
---------------------------------------------------------------------------

activity MB is
  with Mode;

  ports
    p25 : (Signal, BIT_Test.BIT_Assessment, in);
    p26 : (Channel, Mode.number, out);
    p27 : (Signal, Reports.status, out);
  end ports;

  auxiliary definitions
    constant l : Time;
    constant wcet : Time;
    constant u : Time;
    constant t0 : Time;
    constant t1 : Time;
  end auxiliary definitions;

  local state
    missile_state : BIT_Test.BIT_Assessment;
  end local state;

  operations
    op tests (BIT_msg : BIT_Test.BIT_Assessment) m : Mode.number;
     ext write missile_state;
      pre true;
      post true;
    end op;

    op write_report () rep : Reports.status;
     ext read missile_state;
      pre true;
      post true;
    end op;
  end operations;

  states
    static MB_A
      transition goes to do_tests on read p25;
    end state;

    dynamic do_tests
      op tests reads from p25 writes to p26;
      timing [l,wcet,u];
      transition goes to MB_C on true;
    end state;

    static MB_C
      transition goes to write_reports on write p26;
    end state;

    dynamic write_reports
      op write_report writes to p27;
      timing [l,wcet,u];
      transition goes to MB_A on true;
    end state;

    initial MB_A;
  end states;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
end activity;

----------------------------------------------------------------------------
--                                                                        --
-- TG Specification                                                       --
-- ================                                                       --
-- The "Timeout Generator" (TG) activity loops...
----------------------------------------------------------------------------

activity TG is
  with Mode, Clock;

  ports
     p34 : (Stimulus, NULL, in);
     p35 : (Channel, Mode.number, out);
  end ports;

  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
     constant t0 : Time;
     constant t1 : Time;
  end auxiliary definitions;

  operations
     op write_timeout () m : Mode.number;
       pre true;
       post m = "Timeout";
     end op;
  end operations;

  states
--STIM not READ!
     static TG_A
        transition goes to TG_B on read p34;
     end state;

     static TG_B
        transition goes to TG_C on [to1,to2];
     end state;

     dynamic TG_C
        op write_timeout writes to p35;
        timing [l,wcet,u];
        transition goes to TG_D on true;
     end state;

     static TG_D
        transition goes to TG_A on write p35;
     end state;

     initial TG_A;
  end states;

end activity;

----------------------------------------------------------------------------
--                                                                        --
-- IMU_SS Specification                                                   --
--                                                                        --
-- Desc: The "Inertial Measurement Unit" (IMU) Sub-system loops...
----------------------------------------------------------------------------

activity IMU_SS is
  with Image, Clock;

  ports
     pclk1 : (Stimulus, Clock.tick, in);
     p40   : (Signal, Image.Processed_Image, out);
  end ports;
```

**Thesis-caseStudyHostDesign.rtnsl**

```
  auxiliary definitions
     constant l : Time;
     constant wcet : Time;
     constant u : Time;
  end auxiliary definitions;

  operations
     op write_data () output : Image.Processed_Image;
       pre true;
       post true;
     end op;
  end operations;

--STIM not READ!

  states
     static IMU_A
        transition goes to IMU_B on read pclk1;
     end state;

     dynamic IMU_B
        op write_data writes to p40;
        timing [l,wcet,u];
        transition goes to IMU_A on true;
     end state;

     initial IMU_A;
  end states;
end activity;


----------------------------------------------------------------------------
--                                                                        --
-- AC_SS Specification                                                    -
--                                                                        --
-- Desc: The "Actuator" Sub-system loops...
----------------------------------------------------------------------------

activity AC_SS is
  with Image;

  ports
     p36 : (Signal, Image.Raw_Image, out);
  end ports;

  operations
     op write_data () output : Image.Raw_Image;
       pre true;
       post true;
     end op;
  end operations;

  states
     dynamic AC_A
        op write_data writes to p36;
        timing [l,wcet,u];
        transition goes to AC_A on true;
     end state;

     initial AC_A;
  end states;
end activity;


----------------------------------------------------------------------------
--
-- A_SS Specification
--
-- Desc: The "Aircraft" sub-system loops ...
----------------------------------------------------------------------------

activity A_SS is
```

**Thesis–caseStudyHostDesign.rtnsl**

```
  with Actuators;

  ports
     p38 : (Signal, Actuators.cmd, in);
  end ports;

  local state
     xyz : Actuators.pos;
  end local state;

  operations
     op read_data (in_cmd : Actuators.cmd);
      ext write xyz;
       pre true;
       post true;
     end op;
  end operations;

  states
     static A_A
        transition goes to A_B on read p38;
     end state;

     dynamic A_B
        op read_data reads from p38;
        timing [1,wcet,u];
        transition goes to A_A on true;
     end state;

     initial A_A;
  end states;
end activity;


----------------------------
--                        --
-- Begin IDA Descriptions --
--                        --
----------------------------


ida AIM is
   Kind Signal;
   Datatype Image.Raw_Image;
end ida;

ida ME is
   Kind Channel;
   Datatype Mode.number;
end ida;

ida AM is
   Kind Pool;
   Datatype Image.Processed_Image;
end ida;

ida AID is
   Kind Signal;
   Datatype INS_Data.pos;
end ida;

ida BC is
   Kind Signal;
   Datatype BIT_Test.BIT_Assessment;
end ida;

ida MS1 is
   Kind Signal;
   Datatype Missile_state.status;
end ida;

ida MS2 is
   Kind Pool;
```

**Thesis–caseStudyHostDesign.rtnsl**

```
   Datatype Missile_state.pos;
end ida;

ida AD is
   Kind Signal;
   Datatype Actuators.cmd;
end ida;

ida TS is
   Kind Stimulus;
   Datatype NULL;
end ida;


---------------------------
--                       --
-- Begin RTN Descriptios --
--                       --
---------------------------


rtn net1 is
  MC.p1 reads from ME;
  AC_SS.p36 writes to AIM;
  RAM.p13 reads from AIM;
  RAM.p15 writes to ME;
  RAM.p16 writes to AM;
  RAM.p17 writes to AID;
  RAM.p19 writes to BC;
  MB.p25 reads from BC;
--  MB.p26 writes to ME;
  SA.p12 reads from AM;
  SA.p9 reads from MS1;
  SA.p10 writes to AD;
  TA.p20 reads from AID;
  TA.p21 reads from MS2;
--  TA.p22 writes to ME;
  A_SS.p38 reads from AD;
  MC.p0 writes to TS;
  TG.p34 reads from TS;
  INav.p6 writes to MS1;
  INav.p8 writes to MS2;
end rtn;


-------------------------------
--                           --
-- Begin Theory Descriptions --
--                           --
-------------------------------


theory
  miat : Time
  deadline : Time

  Assumption1 : AXIOM
     FORALL (i: Occ, t1, t2: Time):
        th(a1_rds_p1, i, t1) AND th(a1_rds_p1, i+1, t2) IMPLIES
            t2 >= t1 + miat

  Assumption2 : AXIOM
     miat > deadline

  Assumption3 : AXIOM
     deadline > a1_u1 + a2_u2

  Theorem1 : THEOREM
     FORALL (i: Occ, t1: Time):
        th(a1_rds_p1, i, t1) IMPLIES
           EXISTS (t2: Time): th(a2_we_p4, i, t2) AND
               t2 <= t1 + deadline

  Theorem2 : THEOREM
```

**Thesis-caseStudyHostDesign.rtnsl**

```
      FORALL (i: Occ, t1: Time):
         th(a2_we_p4, i, t1) IMPLIES
            EXISTS (t2: Time): th(a1_rds_p1, i, t2) AND t2 < t1
end;

---------
--     --
-- END --
--     --
---------
```

## A.2 Modified Axiomatic Semantics

Those axioms modified for the $TA_a$ activity are:

TA_ax_31_rp1 : AXIOM
$\forall i: Occ, t_1, t_2: Time \cdot \Theta(\uparrow TA_rp1, i, t_1) \wedge \Theta\downarrow TA_rp1, i, t_2 \Rightarrow$
$\neg post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2)) \vee$
$post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2))$

TA_ax_31_rp2 : AXIOM
$\forall i: Occ, t_1, t_2: Time \cdot \Theta(\uparrow TA_rp2, i, t_1) \wedge \Theta\downarrow TA_rp2, i, t_2 \Rightarrow$
$\neg post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2)) \vee$
$post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2))$

TA_ax_31_rp3 : AXIOM
$\forall i: Occ, t_1, t_2: Time \cdot \Theta(\uparrow TA_rp3, i, t_1) \wedge \Theta\downarrow TA_rp3, i, t_2 \Rightarrow$
$\neg post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2)) \vee$
$post_TA_check_missile_alignment(TA_incoming_ms(t_1), TA_input(t_1))(TA_MS(t_2))$

TA_rds_p21_ax : AXIOM
$\forall i: Occ, t: Time \cdot \Theta(rds_p21, i, t) \Leftrightarrow$
$\exists j: Occ \cdot \Theta(\uparrow TA_rp1, j, t) \vee \Theta(\uparrow TA_rp2, j, t) \vee \Theta(\uparrow TA_rp3, j, t)$

TA_ax_17_rp1 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\uparrow TA_rp1, i, t_1) \Rightarrow$
$\exists t_2: Time \cdot t_1 + TA_l \leq t_2 \leq t_1 + TA_u \wedge$
$\Theta(\downarrow TA_rp1, i, t_2)$

TA_ax_17_rp2 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\uparrow TA_rp2, i, t_1) \Rightarrow$
$\exists t_2: Time \cdot t_1 + TA_l \leq t_2 \leq t_1 + TA_u \wedge$
$\Theta(\downarrow TA_rp2, i, t_2)$

TA_ax_17_rp3 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\uparrow TA_rp3, i, t_1) \Rightarrow$
$\exists t_2: Time \cdot t_1 + TA_l \leq t_2 \leq t_1 + TA_u \wedge$
$\Theta(\downarrow TA_rp3, i, t_2)$

TA_ax_5_rp1 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\downarrow TA_rp1, i, t_1) \Rightarrow$
$(\mathsf{true} \wedge \Theta(\uparrow TA_D, i, t_1)) \vee$
$(\exists j: Occ \cdot valueFault(_, j, t_1) \wedge \Theta(\uparrow TA_rp2, j, t_1))$

TA_ax_5_rp2 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\downarrow TA_rp2, i, t_1) \Rightarrow$
$(\mathsf{true} \wedge \Theta(\uparrow TA_D, i, t_1)) \vee$
$(\exists j: Occ \cdot valueFault(_, j, t_1) \wedge \Theta(\uparrow TA_rp3. j. t_1))$

TA_ax_5_rp3 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow TA_rp3, i, t_1) \Rightarrow$
$(\text{true} \wedge \Theta(\Uparrow TA_D, i, t_1))$

TA_ax_4_rp1 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow TA_B, i, t_1) \Rightarrow$
$\text{true} \wedge \Theta(\Uparrow TA_rp1, i, t_1)$

Those axioms modified for the $SA_a$ activity are:

v1_ax_1 : AXIOM
$\forall t_1: Time \cdot \exists i: Occ \cdot \Theta(v1_we_pY, i, t_1) \Leftrightarrow$
$\exists j: Occ \cdot \Theta(\Downarrow v1_stateV, j, t_1)$

v1_ax_2 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow v1_stateV, i, t_1) \Rightarrow$
$\exists t_2: Time \cdot t_2 + l \leq t_1 \leq t_2 + u \wedge \Theta(\Uparrow v1_stateV, i, t_2)$

v1_ax_3 : AXIOM
$\forall i: Occ, t_1, t_2: Time \cdot \Theta(\Uparrow v1_stateV, i, t_1) \wedge \Theta(\Downarrow v1_stateV, i, t_2) \Rightarrow$
$post_vote(t_1, t_2)$

IH_ax_11' : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Uparrow IH_A, i, t_2)$
$\exists j: Occ, t_2: Time \cdot \Theta(stim_clk4, j, t_2) \wedge \Theta(\Downarrow IH_A, i, t_2) \vee$
$t_1 + l \leq t_2 \leq t_1 + u \wedge \Theta(\Downarrow A_A, i, t_2)$

IH_ax_14' : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow IH_A, i, t_1) \Rightarrow$
$\exists j: Occ, t_2: Time \cdot \Theta(stim_clk4, j, t_1) \vee$
$t_1\text{-}u \leq t_2 \leq t_1\text{-}l \wedge \Theta(\Uparrow IH_A, i, t_2)$

IH_ax_2' : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow IH_A, i, t_1) \Rightarrow$
$\exists j, k: Occ \cdot \Theta(\Uparrow IH_abandon_check, j, t_1) \wedge \Theta(stim_clk4, k, t_1) \vee$
$\Theta(\Uparrow IH_watchdog, j, t_1) \wedge \Theta(stim_clk4, k, t_1)$

IH_ax_18 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Uparrow IH_watchdog, i, t_1) \Rightarrow$
$\exists t_2: Time \cdot t_1 + l \leq t_2 \leq t_1 + u \wedge \Theta(\Downarrow IH_watchdog, i, t_2)$

IH_ax_19 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Downarrow IH_watchdog, i, t_1) \Rightarrow \Theta(\Uparrow IH_wait, i, t_1)$

IH_ax_20 : AXIOM
$\forall i: Occ, t_1: Time \cdot \Theta(\Uparrow IH_wait, i, t_1) \Rightarrow$
$\exists j: Occ, t_2: Time \cdot \Theta(stim_clk4, j, t_2) \wedge \Theta(\Downarrow IH_wait, i, t_2) \vee$
$\Theta(stim_pB, j, t_2) \wedge \Theta(\Downarrow IH_wait, i, t_2)$

IH_ax_21 : AXIOM

$\forall i : Occ, t_1 : Time \cdot \Theta(\Downarrow IH_wait, i, t_1) \Rightarrow$

$\exists j, k : Occ, t_2 : Time \cdot \Theta(\Uparrow IH_abandon_check, j, t_2) \land \Theta(stim_clk4, k, t_2) \lor$
$\Theta(\Uparrow IH_lock_firing, j, t_2) \land \Theta(stim_pB, k, t_2)$

# Appendix B

# Axiomatic Soundness Proofs and Lemmas

## B.1 Soundness Conjectures

The axiom schema's which define $\Omega$ are partitioned into the following families: *Start-up*; *State Exit*; *State Entry*; *Progress* and *Stability* axioms, following the state-machine axiomatisation in [Pay02].

Paynter also stated:

> *The complexity of this schema derives from the fact that it defines the relationship between the number of times the source state has been exited, and the number of times the target has been entered. This depends upon the number of transitions into the target state, and the fact that the guard and static semantics together ensure that the source state will have a single exit transition. This information is static, and may be exploited when instantiating the schema for each pair of states which satisfy the guard to determine which clause of the schema applies. Exploiting this information when instantiating the schema is helpful; trivial proofs about the relationship between target and source indexes need not be repeated for each separate state-machine specification.*

We therefore only select the interesting examples from this schema and show the resulting conjectures sound. Where possible, we cover the array of possibilities in different conjectures, instead of repeating the same justification for each proof.

Note, the conditions in the square brackets at the top of the conjectures, are its guards. These are assumed known in the first step of each proof. Several auxiliary functions are used to aid presentation and are taken from the existing axiomatic semantic definitions.

The following four functions define syntactic properties of state-machines, and are used in the guards to the conjectures below.

$$TransitionsOut : State \times Transition\text{-set} \rightarrow Transition\text{-set}$$
$$TransitionsOut(s, ts) \triangleq \{t \mid t \in ts \cdot t.src = s\}$$

$TransitionsIn : State \times Transition\text{-set} \rightarrow Transition\text{-set}$
$TransitionsIn(s, ts) \triangleq \{t \mid t \in ts \cdot t.trg = s\}$

$Sucessors : State \times State\text{-set} \times Transition\text{-set} \rightarrow State\text{-set}$
$Sucessors(s, state, ts) \triangleq \{s' \in states \mid \exists t \in ts \cdot t.src = s \wedge t.trg = s'\}$

$Predecessors : State \times State\text{-set} \times Transition\text{-set} \rightarrow State\text{-set}$
$Predecessors(s, state, ts) \triangleq \{s' \in states \mid \exists t \in ts \cdot t.src = s' \wedge t.trg = s\}$

The following four functions characterise whether a state has a single ($SE$) or multiple ($ME$) entry transitions, and single ($SX$) or multiple ($MX$) exit transitions:

$SE, ME, SX, MX : State \times State\text{-set} \times Transition\text{-set} \rightarrow \mathbb{B}$
$SE(s, state, ts) \triangleq \#Predecessors(s, states, ts) = 1$
$ME(s, state, ts) \triangleq \#Predecessors(s, states, ts) > 1$
$SX(s, state, ts) \triangleq \#Sucessors(s, states, ts) = 1$
$MX(s, state, ts) \triangleq \#Sucessors(s, states, ts) > 1$

### B.1.1 Ax2 :: Static State Exit

$[is_static_state(s),\ s_t \in enabled(s, ts, tr),\ SE(s, states, ts)]$
$P(\pi) \triangleq \forall t_1 \in \mathrm{inds}\,\pi \cdot \downarrow s \in_i \pi(t_1) \Rightarrow \uparrow s_t \in_i \pi(t_1)$

### B.1.2 Ax3 :: Dynamic State Exit

$[s = dynamic,\ s_t \in enabled(s, ts, tr),\ ME(s)]$
$P(\pi) \triangleq \forall t_1 \in \mathrm{inds}\,\pi \cdot \downarrow s \in_i \pi(t_1) \Rightarrow \uparrow s_t \in_j \pi(t_1)$

### B.1.3 Ax4 :: (Initial) State Entry

$[s = initial,\ MX(s, states, ts),\ ME(s, states, ts),\ i > 1,\ s_s \in Predecessors(s, states.ts)]$
$P(\pi) \triangleq \forall t_1 \in \mathrm{inds}\,\pi \cdot \uparrow s \in_i \pi(t_1) \Rightarrow \exists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow s_s \in_j \pi(t_2)$

### B.1.4 Ax5 :: State Entry

$[s \neq initial,\ s_p \in Predecessor(s, ts)]$
$P(\pi) \triangleq \forall t_1 \in \mathrm{inds}\,\pi \cdot \uparrow s \in_i \pi(t_1) \Rightarrow \downarrow s_p \in_i \pi(t_1)$

### B.1.5 Ax6 :: (Dynamic) State Progress

$[s = dynamic]$
$P(\pi) \triangleq \forall t_1 \in \mathrm{inds}\,\pi \cdot \uparrow s \in_i \pi(t_1) \wedge \mathrm{len}\,\pi \geq t_1 + u \Rightarrow \exists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow s \in_i \pi(t_2) \wedge t_2 \leq t_1 + u$

### B.1.6 Ax7 :: (Static) State Progress, Event Transition

$[s = static,\ label = event]$

$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot e \in_i \pi(t_1) \Rightarrow \downarrow s \in_i \pi(t_1)$

### B.1.7 Ax8 :: (Static) State Progress, Timed Transition

$[s = static,\ label = timed]$

$$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot \uparrow s \in_i \pi(t_1) \wedge \mathsf{len}\,\pi \geq t_1 + u \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot \downarrow s \in_i \pi(t_2) \wedge t_2 \leq t_1 + u$$

### B.1.8 Ax9 :: (Dynamic) State Stability

$[s = dynamic]$

$$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot \downarrow s \in_i \pi(t_1) \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot \uparrow s \in_i \pi(t_2) \wedge t_1\text{-}wcrt \leq t_2 \leq t_1\text{-}bcet$$

### B.1.9 Ax10:: (Static) State Stability, Event Transition

$[s = static,\ label = event]$

$$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot \downarrow s \in_i \pi(t_1) \Rightarrow \exists t_2 \in \mathsf{inds}\,\pi \cdot \uparrow s \in_i \pi(t_2) \wedge t_1\text{-}u \leq t_2 \leq t_1\text{-}l$$

### B.1.10 Ax11:: (Static) State Stability, Timed Transition

$[s = static,\ label = timed]$

$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot \downarrow s \in_i \pi(t_1) \Rightarrow e \in \pi(t_1)$

### B.1.11 Ax12:: Stop States

$[s = static]$

$P(\pi) \triangleq \forall t_1 \in \mathsf{inds}\,\pi \cdot \mathsf{term} \in \pi(\mathsf{len}\,\pi) \Rightarrow \neg \downarrow s \in_i \pi(t_1)$

## B.2 Top Level Soundness Proofs

### B.2.1 Ax2

As Ax3

### B.2.2 Ax3

PROOF (AX3)

from $rtn : RTN$; $\sigma : \Sigma$; $\pi : \Pi_{rtn}$;

| | | |
|---|---|---|
| 1 | inds $\pi : \mathbb{N}$-set | inds-form(h3) |
| 2 | from $t : \mathbb{N}_1$; $t \in$ inds $[es]$; $\uparrow s \in [es](t)$ | |
| 2.1 | $\neg \downarrow s \in [es](t)$ | non-zero(2.h3) |
| | infer $t \in$ inds $[es]$ $\downarrow s \in_j [es](t) \Rightarrow \uparrow s_s \in_i [es](t)$ | $\Rightarrow$-I-right-vac(2.1) |
| 3 | $\forall t \in$ inds $[es] \cdot \downarrow s \in_i [es](t) \Rightarrow \uparrow s_s \in_j [es](t)$ | $\forall$-I-set(1,2) |
| 4 | $P([es])$ | folding(3) |
| 5 | from $P(\pi')$; $s : ES$; $inv\text{-}\Pi_{rtn}(\pi' \curvearrowright [s])$ | |
| 5.1 | from $t_1' \in$ inds $\pi' sconc[s]$; $\downarrow s \in_i \pi' \curvearrowright [s](t_1')$ | |
| | infer $\uparrow s_s \in_j \pi' \curvearrowright [s](t_1')$ | exit-enter-succ(5.1.h1) |
| | infer $t_1' \in$ inds $\pi' sconc[s] \cdot \downarrow s \in_i \pi' \curvearrowright [s](t_1') \Rightarrow \uparrow s_s \in_j \pi' \curvearrowright [s](t_1')$ | $\Rightarrow$-I(5.1.h1,5.1) |
| 6 | $\forall t_1' \in$ inds $\pi' sconc[s] \cdot \downarrow s \in_i \pi' \curvearrowright [s](t_1') \Rightarrow \uparrow s_s \in_j \pi' \curvearrowright [s](t_1')$ | $\forall$-I-set(1,5) |
| 7 | $P(\pi' \curvearrowright [s])$ | folding(6) |
| | infer $\forall \pi : \Pi_{rtn} \cdot P(\pi)$ | $\Pi_{rtn}$-indn(4,7) |

■

$$\text{non-zero}\ \frac{t : \mathbb{N}_1;\ \pi : \Pi_{rtn};\quad \uparrow s \in \pi(t)}{\neg \downarrow s \in \pi(t)}$$

$$\text{exit-enter-succ}\ \frac{t : \mathbb{N}_1;\ \sigma : \Sigma;\ \pi : \Pi_{rtn};\quad (mk_tr(s, s_s), \sigma, \pi(1, \ldots, t)) \xrightarrow{tr} \mathbf{true};\quad \downarrow s \in \pi(t)}{\uparrow s_s \in \pi(t)}$$

### B.2.3 Ax4

PROOF (AX4)

| | | |
|---|---|---|
| from $rtn : RTN;\ \sigma : \Sigma;\ \pi : \Pi_{rtn};\ s = initial \ \Rightarrow\ i > 1$ | | |
| 1 | inds $\pi : \mathbb{N}$-set | inds-form(h3) |
| 2 | from $t : \mathbb{N}_1;\ t \in$ inds $[es];\ \uparrow s \in_i [es](t)$ | |
| 2.1 | $i = 1$ | occ-once(2.h2) |
| | infer $\downarrow s_p \in_i [es](t)$ | contradiction(h4,2.1) |
| 3 | $t \in$ inds $[es] \cdot \uparrow s \in_i [es](t) \ \Rightarrow\ \downarrow s_p \in_i [es](t)$ | ⇒-I(2.h2,2) |
| 4 | $\forall t \in$ inds $[es] \cdot \downarrow s \in_i [es](t) \ \Rightarrow\ \uparrow s_s \in_j [es](t)$ | ∀-I-set(1,3) |
| 5 | $P([es])$ | folding(4) |
| 6 | from $P(\pi');\ s : ES;\ inv\text{-}\Pi_{rtn}(\pi' \curvearrowright [s]);\ t' \in$ inds $\pi' \curvearrowright [s]$ | |
| 6.1 | from $\uparrow s \in_i \pi' \curvearrowright [s](t')$ | |
| | infer $\downarrow s_s \in_i \pi' \curvearrowright [s](t')$ | enter-exit-pred(6.1.h1) |
| | infer $\uparrow s \in_i \pi' \curvearrowright [s](t') \ \Rightarrow\ \downarrow s_s \in_i \pi' \curvearrowright [s](t')$ | ⇒-I(6.1.h1,6.1) |
| 7 | $\forall t' \in$ inds $\pi' sconc[s] \cdot \uparrow s \in_i \pi' \curvearrowright [s](t') \ \Rightarrow\ \downarrow s_s \in_i \pi' \curvearrowright [s](t'$ | ∀-I-set(1,6) |
| 8 | $P(\pi' \curvearrowright [s])$ | folding(7) |
| infer $\forall \pi : \Pi_{rtn} \cdot P(\pi)$ | | $\Pi_{rtn}$-indn(5,8) |

■

### B.2.4 Ax5

PROOF (AX5)

| | | |
|---|---|---|
| from $rtn : RTN;\ \sigma : \Sigma;\ \pi : \Pi_{rtn};\ s \neq initial \ \Rightarrow\ i > 1$ | | |
| 1 | inds $\pi : \mathbb{N}$-set | inds-form(h3) |
| 2 | from $t : \mathbb{N}_1;\ t \in$ inds $[es];\ \uparrow s \in_i [es](t)$ | |
| 2.1 | $i = 1$ | occ-once(2.h2) |
| | infer $\downarrow s_p \in_i [es](t)$ | contradiction(h4,2.1) |
| 3 | $t \in$ inds $[es] \cdot \uparrow s \in_i [es](t) \ \Rightarrow\ \downarrow s_p \in_i [es](t)$ | ⇒-I(2.h2,2) |
| 4 | $\forall t \in$ inds $[es] \cdot \downarrow s \in_i [es](t) \ \Rightarrow\ \uparrow s_s \in_j [es](t)$ | ∀-I-set(1,3) |
| 5 | $P([es])$ | folding(4) |
| 6 | from $P(\pi');\ s : ES;\ inv\text{-}\Pi_{rtn}(\pi' \curvearrowright [s]);\ t' \in$ inds $\pi' \curvearrowright [s]$ | |
| 6.1 | from $\uparrow s \in_i \pi' \curvearrowright [s](t')$ | |
| | infer $\downarrow s_s \in_i \pi' \curvearrowright [s](t')$ | enter-exit-pred(6.1.h1) |
| | infer $\uparrow s \in_i \pi' \curvearrowright [s](t') \ \Rightarrow\ \downarrow s_s \in_i \pi' \curvearrowright [s](t')$ | ⇒-I(6.1.h1,6.1) |
| 7 | $\forall t' \in$ inds $\pi' sconc[s] \cdot \uparrow s \in_i \pi' \curvearrowright [s](t') \ \Rightarrow\ \downarrow s_s \in_i \pi' \curvearrowright [s](t'$ | ∀-I-set(1,6) |
| 8 | $P(\pi' \curvearrowright [s])$ | folding(7) |
| infer $\forall \pi : \Pi_{rtn} \cdot P(\pi)$ | | $\Pi_{rtn}$-indn(5,8) |

■

$$\text{occ-once}\ \frac{t, i : \mathbb{N}_1;\ s : ES;\ e : Event;\quad t \in \mathsf{inds}\,[s] \cdot e \in_i [s](t)}{i = 1}$$

$$\text{enter-exit-pred}\ \frac{t : \mathbb{N}_1;\ \sigma : \Sigma;\ \pi : \Pi_{rtn};\quad (mk_tr(s_p, s), \sigma, \pi(1, \ldots, t)) \xrightarrow{tr} \mathbf{true};\quad \updownarrow s \in \pi(t)}{\downarrow s_p \in \pi(t)}$$

### B.2.5 Ax6

done in main body of thesis

### B.2.6 Ax8

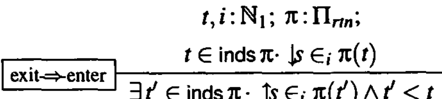
as Ax6

### B.2.7 Ax9

PROOF (AX9)

| | | |
|---|---|---|
| | from $rtn : RTN;\ \sigma : \Sigma;\ \pi : \Pi_{rtn}$ | |
| 1 | $\text{inds}\,\pi : \mathbb{N}_1\text{-set}$ | inds-form(h3) |
| 2 | $[es] : \Pi_{rtn}$ | ?-form |
| 3 | from $t_1 \in \text{inds}\,[es];\ \downarrow s \in_i [es](t_1)$ | |
| 3.1 | $\exists t'' \in \text{inds}\,[es] \cdot \uparrow s \in_i [es](t'') \wedge t'' < t_1$ | exit-⇒-enter() |
| 3.2 | $\neg \downarrow s \in_i [es](t_1)$ | |
| | infer $\exists t_2 \in \text{inds}\,[es] \cdot \uparrow s \in_i [es](t_2) \wedge t_1\text{-}wcrt \leq t_2 \leq t_1\text{-}bcet$ | contradiction() |
| 4 | $t_1 \in \text{inds}\,[es] \cdot \downarrow s \in_i [es](t_1) \Rightarrow \exists t_2 \in \text{inds}\,[es] \cdot \uparrow s \in_i [es](t_2) \wedge t_1\text{-}wcrt \leq t_2 \leq t_1\text{-}bcet$ | ⇒-I(3.h1,3.h2,3) |
| 5 | $\forall t_1 \in \text{inds}\,[es] \cdot \downarrow s \in_i [es](t_1) \Rightarrow \exists t_2 \in \text{inds}\,[es] \cdot \uparrow s \in_i [es](t_2) \wedge t_1\text{-}wcrt \leq t_2 \leq t_1\text{-}bcet$ | ∀-I-set(1,4) |
| 6 | $P([es])$ | folding(5) |
| 7 | from $P(\pi');\ s : ES;\ inv\text{-}\Pi_{rtn}(\pi' \frown [s]);$ $t'_1 \in \text{inds}\,\pi' \frown [s];\ \downarrow s \in_i \pi' \frown [s](t'_1)$ | |
| 7.1 | $\exists t'' \in \text{inds}\,\pi' \frown [s] \cdot \uparrow s \in_i \pi' \frown [s](t'') \wedge t'' < t'_1$ | exit-⇒-enter(7.h4,7.h5) |
| 7.2 | from $t'_2 \in \text{inds}\,\pi \frown [s];\ \uparrow s \in_i \pi' \frown [s](t'_2) \wedge t'_2 < t'_1$ | |
| 7.2.1 | $t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | prem-??? |
| | infer $\uparrow s \in_i \pi' \frown [s](t'_2) \wedge t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | ∧-I(7.2.h2,7.2.1) |
| 7.3 | $\uparrow s \in_i \pi' \frown [s](t'_2) \wedge t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | ∃-E(7.1,7.2.h1,7.2.h2,7.2) |
| | infer $\exists t'_2 \in \text{inds}\,\pi' \frown [s] \cdot \uparrow s \in_i \pi' \frown [s](t'_2) \wedge t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | ∃-I-set(7.3) |
| 8 | $t'_1 \in \text{inds}\,\pi' \frown [s] \cdot \downarrow s \in_i \pi' \frown [s](t'_1) \Rightarrow \exists t'_2 \in \text{inds}\,\pi \frown [s] \cdot \uparrow s \in_i \pi \frown [s](t'_2) \wedge t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | ⇒-I(7.h4,7.h5,7) |
| 9 | $\forall t'_1 \in \text{inds}\,\pi' \frown [s] \cdot \downarrow s \in_i \pi' \frown [s](t'_1) \Rightarrow \exists t'_2 \in \text{inds}\,\pi \frown [s] \cdot \uparrow s \in_i \pi \frown [s](t'_2) \wedge t'_1\text{-}wcrt \leq t'_2 \leq t'_1\text{-}bcet$ | ∀-I-set(7.h4,8) |
| 10 | $P(\pi' \frown [s])$ | folding(9) |
| | infer $\forall \pi : \Pi_{rtn} \cdot P(\pi)$ | $\Pi_{rtn}$-indn(6,10) |

■



$$\frac{t, i : \mathbb{N}_1;\ \pi : \Pi_{rtn};\quad t \in \text{inds}\,\pi \cdot \downarrow s \in_i \pi(t)}{\exists t' \in \text{inds}\,\pi \cdot \uparrow s \in_i \pi(t') \wedge t' < t} \quad \text{(exit-⇒-enter)}$$

### B.2.8 Ax10

as Ax9

## B.3 Additional Lemmas

$$\frac{a : A, s : A^*}{\text{len}\, s \frown [a] = succ(\text{len}\, s)} \quad \text{(len-}\frown\text{)}$$

PROOF (LEN-$\curvearrowright$)

from $a : A, s : A^*$
infer $\text{len}\, s \curvearrowright [a] = succ(\text{len}\, s)$

■

PROOF (=-∨-{}-I)

from $a : \mathbb{N};\ b : \mathbb{N};\ a = b$

| | | |
|---|---|---|
| 1 | $0 : \mathbb{N}$ | 0-form |
| 2 | $a + 0 = a$ | +-defn-0-right(h1) |
| 3 | $a + 0 = b$ | =-subs-right(b)(h1,h3,2) |
| 4 | $\exists k : \mathbb{N} \cdot a + k = b$ | ∃-I(1,3) |
| infer | $a \geq b$ | folding(4) |

■

$$\boxed{\text{=-∨-\{\}-I}}\ \frac{s : A\text{-set}}{s = \{\} \vee s \neq \{\}}$$

PROOF (=-∨-{}-I)

from $s : A$-set

| | | |
|---|---|---|
| 1 | $\{\} : A$-set | {}-form |
| 2 | $\delta(s = \{\})$ | δ-=-I(h1,1) |
| infer | $s = \{\} \vee s \neq \{\}$ | unfolding(2) |

■

$$\boxed{\text{=-∨-}\geq\text{-I}}\ \frac{a : \mathbb{N}, b : \mathbb{N}, a = b}{a \geq b}$$

# Appendix C

# VDM Tool support for SOS definitions

In this Appendix, we present the full VDM-SL model which defines the abstract syntax on which our SOS rules are specified. In an attempt to syntax and type check these rules we specify each SOS rule as an implicit VDM-SL functions [ABV94] and examine the implementation considerations that a concurrent, non-deterministic model pose.

## C.1 Abstract Syntax

types

17.0 $Id$ = token;

18.0 $Var$ = token;

19.0 $Expr$ = token;

20.0 $Event$ = token;

21.0 $Fault$ = LATE_EXIT;

22.0 $Bounds :: bcet : \mathbb{N}$
.1 $bcrt : \mathbb{N}$
.2 $wcet : \mathbb{N}$;

23.0 $Port :: id : Id$
.1 $dir$ : IN | OUT;

24.0 $TimeBound :: lower : \mathbb{N}$
.1 $upper : \mathbb{N}$;

25.0 *Label* = *TimeBound* | *Event* | *Expr* | *Fault*;

26.0 *Static-State* :: *id* : *Id*;

27.0 *Operation* :: *id* : *Id*
.1 *read-vars* : *Var*-set
.2 *written-vars* : *Var*-set
.3 *input-parameter* : [*Id*]
.4 *output-result* : [*Id*]
.5 *preC* : *Expr*
.6 *postC* : *Expr*;

28.0 *Dynamic-State* :: *id* : *Id*
.1 *ops* : *Operation*-set
.2 *bounds* : *Bounds*;

29.0 *State* = *Static-State* | *Dynamic-State*;

30.0 *Transition* :: *src* : *Id*
.1 *trg* : *Id*
.2 *l* : *Label*;

31.0 *SM* :: *ss* : *State*-set
.1 *ts* : *Transition*-set
.2 *initial* : *Id*;

32.0 *Activity* :: *input-ports* : *Port*-set
.1 *output-ports* : *Port*-set
.2 *local-state* : $Id \xrightarrow{m} Var$
.3 *ops* : *Operation*-set
.4 *sm* : *SM*;

33.0 *IDA* :: *id* : *Id*
.1 *size* : $\mathbb{N}$
.2 *protocol* : POOL | CHAN | SIGNAL;

34.0 *Connection* :: *act-id* : *Id*
.1 *port-id* : *Id*
.2 *ida-id* : *Id*
.3 *dir* : IN | OUT;

35.0 *rtn-spec* = token;

36.0 $RTN :: acts : Activity\text{-set}$
.1 $idas : IDA\text{-set}$
.2 $net : rtn\text{-}spec;$

## C.2 Context Conditions

### C.2.1 Auxiliary Objects

We need an object to record the static information about the given real time network:

37.0 $RTN\text{-}Types = Id \xrightarrow{m} (Activity \mid IDA \mid Connection);$

## C.3 (Normative) Semantics

What follows are two presentations of (hopefully) the same semantics. In our attempt to define an SOS semantics in the Plotkin Style, we give side-by-side to each rule a VDM function which (in the least) syntax & type checks our rules.

### C.3.1 Semantic Objects

Dynamic information[1] about RTN components are stored in:

38.0 $= Id \xrightarrow{m} (port \mid ida \mid act \mid dySt);$

and the history of a RTN

39.0 $= (Event\text{-set})^{*};$

The (dynamic) RTN components we consider are:

40.0 $port = Var;$

41.0 $ida :: ls : Var^{*};$

42.0 $act :: cs : Id$
.1 $ls : Id \xrightarrow{m} Var$
.2 $status :$ INIT | WAIT_RD | WAIT_WE | WAIT_TR | INSS | INDS | TERM | FAULT;

43.0 $dySt :: ops : Id^{*}$

Note, the $\mathrm{dom}\,\Sigma \subset \mathrm{dom}\, RTN\text{-}Types$ in that for all components with dynamic properties (port, IDAs, Activities & Dynamic States) in *RTN-Types* their dynamic properties are in $\Sigma$. It is envisaged some initialisation rule[2] would define this.

[1] Static information can be obtained from *RTN-Types*
[2] It is not clear currently what this would look like

## C.3.2 Auxiliary Functions

*insf*() determines whether an *id* of an Activity, *A* component is a state.

functions

44.0 $insf : Activity \times Id \rightarrow \text{INSS} \mid \text{INDS}$

.1 $insf(A, id) \triangleq$

.2 let $s \in A.sm.ss$ be st

.3 $s.id = id$ in

.4 if is_$(s, Dynamic\text{-}State)$

.5 then INDS

.6 else INSS;

The following three auxiliary functions are required to convert a set of Operation to a sequence of Operation ids...

45.0 $getOpIds : Operation\text{-set} \rightarrow Id^*$

.1 $getOpIds(opset) \triangleq$

.2 let $ops = set2seq[Operation](opset)$ in

.3 $[ops(i).id \mid i \in \text{inds } ops]$;

46.0 $set2seq[@elem] : (@elem\text{-set}) \rightarrow (@elem^*)$

.1 $set2seq(s) \triangleq$

.2 $seqBuilder[@elem](s, [])$;

47.0 $seqBuilder[@elem] : (@elem\text{-set}) \times (@elem^*) \rightarrow (@elem^*)$

.1 $seqBuilder(s, sq) \triangleq$

.2 if $s = \{\}$

.3 then $sq$

.4 else let $x \in s$ in

.5 $seqBuilder[@elem](s \setminus \{x\}, [x] \frown sq)$;

## C.3.3 Semantic Rules

The types of the required relations are

$$\xrightarrow{s} : ((RTN\text{-}Types \times \Sigma \times \Pi) \times (\Sigma \times \Pi))$$

Other semantic (transition) relations defined are:

$$\xrightarrow{rd} : ((RTN\text{-}Types \times \Sigma \times \Pi) \times (\Sigma \times \Pi))$$

$\xrightarrow{we} : ((RTN\text{-}Types \times \Sigma \times \Pi) \times (\Sigma \times \Pi))$

$\xrightarrow{rd} : ((Port \times \Sigma) \times Val)$

$\xrightarrow{e} : ((Expr \times \Sigma) \times \mathbb{B})$

The relations above will form the function signatures for the VDM functions, for example:

48.0 $\xrightarrow{s} : RTN\text{-}Types \times \Sigma \times \Pi \times \Sigma \times \Pi \rightarrow \mathbb{B}$

#### C.3.3.1 Auxiliary Semantic Functions

49.0 $expr : Expr \times Id \xrightarrow{m} Var \rightarrow \mathbb{B}$

.1 $expr(e, ls) \triangleq$

.2 is not yet specified;

### C.3.3.2 Operations

The meaning of a single operation is defined with four rules, depending on whether the operation reads, writes, does both or neither:

**Reads, not Write**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p \neq \mathbf{nil}, out_p = \mathbf{nil} \\ (in_p, RTN, \sigma_1) \xrightarrow{rd} v \\ ls' = ls \dagger \{in_p \mapsto v\} \\ (pre, ls) \xrightarrow{e} \mathbf{true} \\ (post, ls') \xrightarrow{e} \mathbf{true} \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathrm{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls', \mathrm{INDS})\}, \{rd(in_p)\})} \quad \text{dySt-op-rd}$$

VDM Function:

50.0 $dySt\text{-}op\text{-}rd(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in \mathrm{dom}$ be st $\mathrm{is_}((a), act)$ in

.2 let mk-$act(cs, ls, \mathrm{INDS}) = (a)$ in

.3 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.4 $(cs).ops \neq [] \wedge$

.5 hd $(cs).ops = id \wedge$

.6 $in\text{-}p \neq \mathrm{nil} \wedge$

.7 $out\text{-}p = \mathrm{nil}$ in

.8 let $inp \in RTN(a).input\text{-}ports$ be st

.9 $inp.id = in\text{-}p$ in

.10 let $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp, )\}$ in

.11 $expr(preC, ls) \wedge expr(postC, ls')$

.12 post let $a \in$ dom be st
.13 $((a).cs).ops \neq []$ in
.14 let $op \in RTN(a).ops$ be st
.15 hd $((a).cs).ops = op.id \wedge$
.16 $op.input\text{-}parameter \neq$ nil in
.17 let $inp \in RTN(a).input\text{-}ports$ be st
.18 $inp.id = op.input\text{-}parameter$ in
.19 let $ls' : Id \xrightarrow{m} Var$ be st
.20 $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp,)\} \wedge$
.21 $expr(op.preC, (a).ls) \wedge$
.22 $expr(op.postC, ls')$ in
.23 let mk-$act(cs, ls,$ INDS$) = (a)$ in
.24 $= \dagger\{cs \mapsto$ tl $(cs).ops, a \mapsto$ mk-$act(cs, ls',$ INDS$)\} \wedge$
.25 $= \frown[\{$mk-token $("rd(in_p)")\}]$ ;

**Reads and Writes**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p \neq \mathbf{nil}, out_p \neq \mathbf{nil} \\ (in_p, RTN, \sigma_1) \xrightarrow{rd} v \\ ls' = ls \dagger \{in_p \mapsto v\} \\ (pre, ls) \xrightarrow{e} \mathbf{true} \\ (out_p, v', ls') \xrightarrow{we} ls'' \\ (post, ls'') \xrightarrow{e} \mathbf{true} \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathrm{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls'', \mathrm{INDS})\}, \{rd(in_p), we(out_p)\})} \text{ dySt-op-rdwe}$$

51.0 $dySt\text{-}op\text{-}rdwe\,(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st is_$((a), act)$ in

.2 let mk-$act\,(cs, ls, \mathrm{INDS}) = (a)$ in

.3 let mk-$Operation\,(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN\,(a).ops$ be st

.4 $(cs).ops \neq [] \wedge$

.5 hd $(cs).ops = id \wedge$

.6 $in\text{-}p \neq$ nil $\wedge$

.7 $out\text{-}p \neq$ nil in

.8 let $inp \in RTN\,(a).input\text{-}ports$ be st

.9 $inp.id = in\text{-}p$ in

.10 let $ls' = (a).ls \dagger \{inp.id \mapsto rdport\,(inp, )\}$ in

.11 $expr\,(preC, ls) \wedge$

.12 let $outp \in RTN\,(a).output\text{-}ports$ be st

.13 $outp.id = out\text{-}p$ in

.14 let $ls'' : Id \xrightarrow{m} Var$ in

.15 $\exists v : Var \cdot$

.16 $ls'' = (a).ls \dagger ls' \dagger \{outp.id \mapsto weport\,(outp, v, )\} \wedge$

.17 $expr\,(postC, ls'')$

.18 post let $a \in$ dom be st
.19 $((a).cs).ops \neq []$ in
.20 let $op \in RTN(a).ops$ be st
.21 hd $((a).cs).ops = op.id \wedge$
.22 $op.input\text{-}parameter \neq$ nil $\wedge$
.23 $op.output\text{-}result \neq$ nil in
.24 let $inp \in RTN(a).input\text{-}ports$ be st
.25 $inp.id = op.input\text{-}parameter$ in
.26 let $ls' : Id \xrightarrow{m} Var$ be st
.27 $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp,)\} \wedge$
.28 $expr(op.preC, (a).ls)$ in
.29 let $outp \in RTN(a).output\text{-}ports$ be st
.30 $outp.id = op.output\text{-}result$ in
.31 let $ls'' : Id \xrightarrow{m} Var$ be st
.32 $\exists v : Var \cdot$
.33 $ls'' = (a).ls \dagger ls' \dagger \{outp.id \mapsto weport(outp, v,)\} \wedge$
.34 $expr(op.postC, ls'')$ in
.35 let mk-$act(cs, ls, \text{INDS}) = (a)$ in
.36 $= \dagger\{cs \mapsto$ tl $(cs).ops, a \mapsto$ mk-$act(cs, ls'', \text{INDS})\} \wedge$
.37 $= \curvearrowright[\{$mk-token ("$rd(in_p)$"), mk-token ("$we(out_p)$")$\}]$ ;

### No Read, but Write

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p = \mathbf{nil}, out_p \neq \mathbf{nil} \\ (out_p, v', ls) \xrightarrow{we} ls' \\ (post, ls') \xrightarrow{e} \mathbf{true} \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathrm{tl}\,\sigma_1(cs).ops, a \mapsto mk_act(cs, ls', \mathrm{INDS})\}, \{we(out_p)\})} \quad \boxed{\text{dySt-op-we}}$$

52.0 $dySt\text{-}op\text{-}we(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st is_$((a), act)$ in

.2 let mk-$act(cs, ls, \mathrm{INDS}) = (a)$ in

.3 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.4 $(cs).ops \neq [] \wedge$

.5 hd $(cs).ops = id \wedge$

.6 $in\text{-}p =$ nil $\wedge$

.7 $out\text{-}p \neq$ nil in

.8 let $outp \in RTN(a).output\text{-}ports$ be st

.9 $outp.id = out\text{-}p$ in

.10 let $ls' : Id \xrightarrow{m} Var$ in

.11 $\exists v : Var \cdot$

.12 $ls' = (a).ls \dagger \{outp.id \mapsto weport(outp, v, )\} \wedge$

.13 $expr(postC, ls')$

.14 post let $a \in$ dom be st

.15 $((a).cs).ops \neq []$ in

.16 let $op \in RTN(a).ops$ be st

.17 hd $((a).cs).ops = op.id \wedge$

.18 $op.input\text{-}parameter =$ nil $\wedge$

.19 $op.output\text{-}result \neq$ nil in

.20 let $outp \in RTN(a).output\text{-}ports$ be st

.21 $outp.id = op.output\text{-}result$ in

.22 let $ls' : Id \xrightarrow{m} Var$ be st

.23 $\exists v : Var \cdot$

.24 $ls' = (a).ls \dagger \{outp.id \mapsto weport(outp, v.)\} \wedge$

.25 $expr(op.postC, ls')$ in

.26 let mk-$act(cs, ls, \mathrm{INDS}) = (a)$ in

.27 $= \dagger\{cs \mapsto$ tl $(cs).ops, a \mapsto$ mk-$act(cs, ls', \mathrm{INDS})\} \wedge$

.28 $= \curvearrowright[\{$mk-token ("$we(out_p)$")$\}]$ ;

### Neither Reads, nor Write

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [\,] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\, \sigma_1(cs).ops = id \\ in_p = \mathbf{nil}, out_p = \mathbf{nil} \\ (pre, ls) \xrightarrow{e} \mathbf{true} \\ (post, ls) \xrightarrow{e} \mathbf{true} \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{cs \mapsto \mathrm{tl}\, \sigma_1(cs).ops\}, \{\})} \; \boxed{\text{dySt-op}}$$

```
53.0  dySt-op (RTN : RTN-Types, :, :, :, :) r : 𝔹
 .1   pre let a ∈ dom  be st is_( (a), act) in
 .2       let mk-act (cs, ls, INDS) = (a) in
 .3       let mk-Operation (id, read, we, in-p, out-p, preC, postC) ∈ RTN (a).ops in
 .4       (cs).ops ≠ [] ∧
 .5       hd (cs).ops = id ∧
 .6       in-p = nil ∧
 .7       out-p = nil
 .8   post let a ∈ dom  be st
 .9                  ( (a).cs).ops ≠ [] in
 .10       let op ∈ RTN (a).ops be st
 .11                 hd ( (a).cs).ops = op.id ∧
 .12                 op.input-parameter = nil ∧
 .13                 op.output-result = nil in
 .14       let mk-act (cs, ls, INDS) = (a) in
 .15        = †{cs ↦ tl (cs).ops, a ↦ mk-act (cs, ls, INDS)} ∧
 .16        = ⁀[{}] ;
```

#### C.3.3.3 States

#### Initial States

Associated operations and events of states are evaluated once a state is entered, therefore the semantics for the *initial state* are simply to *enter* the state.

**Static State**

Plotkin Rule:

$$\frac{a \in \mathrm{dom}\,\sigma_1 \qquad \sigma_1(a) = mk_act(_,_,\mathrm{INIT}) \qquad uact = mk_act(RTN(a).sm.initial, RTN(a).local\text{-}state, \mathrm{WAIT_TR})}{(RTN,\sigma_1,\pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\uparrow RTN(a).sm.initial\})}\ \boxed{\text{stSt-init}}$$

VDM Function:

```
54.0  InitSSt(RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom  in
 .2       is_( (a), act) ∧
 .3       (a).status = INIT
 .4   post let a ∈ dom  be st
 .5                 is_( (a), act) ∧
 .6                 (a).status = INIT in
 .7          let uact = mk-act (RTN (a).sm.initial,
 .8                             RTN (a).local-state,
 .9                             WAIT_TR) in
 .10         = †{a ↦ uact} ∧
 .11         = ⌒[mk-token ("up_RTN(a).sm.initial")] ;
```

### Dynamic State

Plotkin Rule:

$$\text{[dySt-init]}\ \frac{\begin{array}{l} a \in \mathrm{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INIT}) \\ uact = mk_act(RTN(a).sm.initial, RTN(a).local\text{-}state, \text{INDS}) \\ mk_Dynamic_State(id, ops, _) \in RTN(a).sm.ss \wedge id = cs \\ trgs = mk_dySt([op \mid op \in ops]) \end{array}}{\begin{array}{l}(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, RTN(a).sm.initial \mapsto trgs\}, \\ \qquad \{\ \uparrow RTN(a).sm.initial\})\end{array}}$$

VDM Function:

```
55.0  InitDySt(RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom in
 .2       is_( (a), act) ∧
 .3       (a).status = INIT
 .4   post let  a ∈ dom  be st
 .5               is_( (a), act) ∧
 .6               (a).status = INIT in
 .7        let uact = mk-act (RTN (a).sm.initial,
 .8                           RTN (a).local-state,
 .9                           INDS) in
 .10        = †{a ↦ uact} ∧
 .11        = ⁀[mk-token ("up_RTN(a).sm.initial")] ;
```

### Static States

Static states are effectively wait states, therefore the semantics simply evaluate an exit transition.

### Dynamic States

A dynamic state may optionally read an input value, and/or write a result to an associated port as specified in the abstract syntax. This behaviour is dealt with at the operation level, rather here we define the dynamic behaviour as such:

Plotkin Rule:

$$\boxed{\text{dySt}}\ \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ \sigma_1(cs).ops = \{\} \\ uact = mk_act(cs, ls, \text{WAIT_TR}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\})}$$

VDM Function:

| | |
|---|---|
| 56.0 | $DynSt(RTN : RTN\text{-}Types, :, :, :)\ r : \mathbb{B}$ |
| .1 | pre let $a \in$ dom in |
| .2 | is_( $(a), act) \wedge$ |
| .3 | $((a).cs).ops = []$ |
| .4 | post let $a \in$ dom be st |
| .5 | is_( $(a), act) \wedge$ |
| .6 | $((a).cs).ops = []$ in |
| .7 | let $= \dagger \{a \mapsto$ mk-$act((a).cs, (a).ls,$ WAIT_TR$)\}$ in |
| .8 | let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss$ be st |
| .9 | $id = (a).cs$ in |
| .10 | $strans(RTN, , , ,) \wedge$ |
| .11 | $\exists t1, t2 \in$ inds $\cdot$ |
| .12 | $t1 + bcet \leq t2 \wedge$ |
| .13 | $t2 \leq t1 + wcet \wedge$ |
| .14 | mk-token ("$up_cs$") $\in (t1) \wedge$ |
| .15 | mk-token ("$dwn_cs$") $\in (t2)$ ; |

### Terminate State

Plotkin Rule:

$$\boxed{\text{term}}\ \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ is_(\sigma_1(a), act) \\ \nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(\sigma_1(a).cs, \sigma_1(a).ls, \mathrm{TERM})\}, \{\})}$$

VDM Function:

57.0 $Term(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st is_( $(a), act)$ in

.2 $\quad \neg \exists tr \in RTN(a).sm.ts \cdot tr.src = (a).cs$

.3 post let $a \in$ dom be st

.4 $\qquad$ is_( $(a), act) \wedge$

.5 $\qquad (\neg \exists tr \in RTN(a).sm.ts \cdot tr.src = (a).cs)$ in

.6 $\quad$ let mk-$act(cs, ls, \mathrm{INDS}) = (a)$ in

.7 $\quad = \dagger\{a \mapsto$ mk-$act(cs, ls, \mathrm{TERM})\} \wedge$

.8 $\quad = \frown [\{\}]$ ;

#### C.3.3.4 Transitions

A VDM 'header' function for the distinction below is:

58.0 $tr\text{-}trans(RTN : RTN\text{-}Types, :, :, :)\ r : \mathbb{B}$

.1 post $str\text{-}trans(RTN, , , ,) \vee dtr\text{-}trans(RTN, , , ,)$ ;

Depending on whether the target state is *static*, or *dynamic*:

**For Dynamic**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathsf{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{WAIT_TR}) \\ tr \in RTN(a).sm.ts \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ mk_Dynamic_State(id, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\, \pi_1 \cdot \Uparrow cs \in_i \pi_1(t_1) \wedge t_1 + bcet \leq \mathsf{len}\, \pi_1 < t_1 + wcet \\ \qquad \nexists t_2 \in \mathsf{inds}\, \pi_1 \cdot t_2 \geq t_1 \wedge \Uparrow cs \in_{i+1} \pi_1(t_2) \\ ins(\sigma_1(tr.trg)) = \text{INDS} \\ uact = mk_act(tr.trg, ls, \text{INDS}) \\ trgs = mk_dySt([op \mid op \in s.ops \cdot s \in RTN(a).sm.ss \wedge s.id = tr.trg]) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact, tr.trg \mapsto trgs\}, \{\Downarrow tr.src, \Uparrow tr.trg\})}\ \text{[tr-dySt]}$$

VDM Function:

59.0 $dtr\text{-}trans(RTN : RTN\text{-}Types, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom in

.2 let mk-$act(cs, ls, \text{WAIT_TR}) = (a)$ in

.3 $insf(RTN(a), cs) = \text{INDS} \wedge$

.4 $\exists tr \in RTN(a).sm.ts \cdot str(tr, ,)$

.5 post let $a \in$ dom be st

.6 mk-$act((a).cs, (a).ls, \text{WAIT_TR}) = (a) \wedge$

.7 $insf(RTN(a), (a).cs) = \text{INDS}$ in

.8 let $tr \in RTN(a).sm.ts$ be st $str(tr, ,)$ in

.9 let $uact =$ mk-$act(tr.trg, (a).ls, \text{INDS})$ in

.10 let $s \in RTN(a).sm.ss$ be st $s.id = tr.trg$ in

.11 let $trgs =$ mk-$dySt(getOpIds(s.ops))$ in

.12 $= \dagger\{a \mapsto uact, tr.trg \mapsto trgs\} \wedge$

.13 $= \curvearrowright[\{$mk-token("$dwn_tr.src$"), mk-token("$up_tr.trg$")$\}]$ ;

**For Static**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\ tr \in RTN(a).sm.ts \\ (tr, \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true} \\ mk_Dynamic_State(id,, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow cs \in_i \pi_1(t_1) \wedge \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot t_2 \geq t_1 \wedge \uparrow cs \in_{i+1} \pi_1(t_2) \wedge \\ \qquad t_1 + bcet \leq \mathsf{len}\,\pi_1 < t_1 + wcet \\ ins(\sigma_1(tr.trg)) = \mathrm{INSS} \\ uact = mk_act(tr.trg, ls, \mathrm{WAIT_TR}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto uact\}, \{\downarrow tr.src, \uparrow tr.trg\})} \quad \text{tr-stSt}$$

VDM Function:

```
60.0  str-trans (RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom in
 .2       let mk-act (cs, ls, WAIT_TR) = (a) in
 .3       insf (RTN (a), cs) = INSS ∧
 .4       ∃ tr ∈ RTN (a).sm.ts · str (tr, , )
 .5   post let a ∈ dom be st
 .6                mk-act ( (a).cs, (a).ls, WAIT_TR) = (a) ∧
 .7                insf (RTN (a), (a).cs) = INSS in
 .8         let tr ∈ RTN (a).sm.ts be st str (tr, , ) in
 .9         let uact = mk-act (tr.trg, (a).ls, WAIT_TR) in
 .10         = † {a ↦ uact} ∧
 .11         = ⁀[{mk-token ("dwn_tr.src"), mk-token ("up_tr.trg")}] ;
```

Three rules totally define the semantics of a transition for the rule:

$$\xrightarrow{e} : ((Transition \times \Sigma \times \Pi) \times \mathbb{B})$$

VDM 'header' function for this relation is:

```
61.0  str (tr : Transition, :, :) r : B
 .1   post tt-tr (tr, , ) ∨ ev-tr (tr, , ) ∨ expr-tr (tr, , ) ;
```

### Time Triggered Transition

Plotkin Rule:

$$\frac{\begin{array}{l} is_(tr.l, TimeBound) \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow src \in_i \pi_1(t_1) \wedge l.lower \leq \mathsf{len}\,\pi_1\text{-}t_1 \leq l.upper \\ \nexists t_2 \in \mathsf{inds}\,\pi_1 \cdot \downarrow trg \in_i \pi_1(t_2) \wedge l.upper \leq \mathsf{len}\,\pi_1\text{-}t_2 \leq l.lower \end{array}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \mathbf{true}} \quad \text{tt-tr}$$

VDM Function:

62.0 *tt-tr* (*tr* : *Transition*, :, :) *r* : $\mathbb{B}$
.1 pre is_(*tr.l*, *TimeBound*) ∧
.2 (∃*t1* ∈ inds ·
.3 mk-token ("*up_tr.src*") ∈ (*t1*) ∧
.4 *tr.l.lower* ≤ (len ) − *t1* ∧
.5 (len ) − *t1* ≤ *tr.l.upper*) ∧
.6 (¬∃*t2* ∈ inds ·
.7 mk-token ("*dwn_tr.trg*") ∈ (*t2*) ∧
.8 *tr.l.lower* ≤ (len ) − *t2* ∧
.9 (len ) − *t2* ≤ *tr.l.upper*)
.10 post true ;

**Event Triggered Transition**

Plotkin Rule:

$$\text{[ev-tr]}\ \frac{is_(tr.l, Event) \quad \exists t_1, t_2 \in \text{inds}\, \pi_1 \cdot \uparrow\!src \in_i \pi_1(t_1) \wedge tr.l \in \pi_1(t_2) \wedge t_2 \geq t_1}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \textbf{true}}$$

VDM Function:

63.0 *ev-tr* (*tr* : *Transition*, :, :) *r* : $\mathbb{B}$
.1 pre is_(*tr.l*, *Event*) ∧
.2 ∃*t1*, *t2* ∈ inds ·
.3 mk-token ("*up_tr.src*") ∈ (*t1*) ∧
.4 *tr.l* ∈ (*t2*) ∧
.5 *t2* ≥ *t1*
.6 post true ;

**Expression Triggered Transition**

Plotkin Rule:

$$\text{[expr-tr]}\ \frac{is_(l, Expression) \quad (l, \sigma_1) \xrightarrow{e} \textbf{true}}{(mk_transition(src, trg, l), \sigma_1, \pi_1) \xrightarrow{tr} \textbf{true}}$$

VDM Function:

64.0 *expr-tr* (*tr* : *Transition*, :, :) *r* : $\mathbb{B}$
.1 pre is_(*tr.l*, *Expr*) ∧
.2 *expr* (*tr.l*, )
.3 post true ;

### C.3.3.5 Ports

In light of defining the *operation* semantic rules, it is clear that we must address the semantics of ports. Ports are the interface between Activities and IDAs. An IDA has two ports.

We must give the semantics here for an activity reading & writing to a port in terms of the local variables used with an operation definition and the value of the port. Later, we must ensure (consistency condition?) that the time and values written and read are consistent with those with the IDA.

Plotkin Rule:

$$\text{rd-Port}\ \frac{dir = \mathrm{IN} \quad \exists i \in \mathrm{dom}\, \sigma \cdot \sigma(i) = id}{(mk_Port(id, dir), \sigma) \xrightarrow{rd} \sigma(i)}$$

VDM Function:

65.0 $rdport(p: Port, :)\ r: Var$

.1 pre $p.dir = \mathrm{IN} \wedge$

.2 $\exists i \in \mathrm{dom} \cdot i = p.id$

.3 post let $i \in \mathrm{dom}$ be st $i = p.id \wedge$

.4 $p.dir = \mathrm{IN}$ in

.5 $r = (i)$ ;

Plotkin Rule:

$$\text{we-Port}\ \frac{\sigma(p).dir = \mathrm{OUT}}{(p, v, \sigma) \xrightarrow{we} \sigma \dagger \{p \mapsto v\}}$$

VDM Function:

66.0 $weport(p: Port, v: Var, :) :$

.1 pre $p.dir = \mathrm{OUT}$

.2 post $= \dagger\{p.id \mapsto v\}$ ;

### C.3.3.6 IDAs

**Pool**

**Read**

Plotkin Rule:

$$\frac{\begin{array}{l} a,c,i \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(i) = mk_ida(vals) \\ RTN(i).protocol = \mathrm{POOL} \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_RD}) \\ s \in RTN(a).sm.ss \\ mk_op(id, rd, we, in_p, out_p, pre, post) \in s.ops \wedge \mathrm{hd}\,\sigma(cs).ops = id \\ in_p \neq \mathbf{nil} \\ RTN(c) = mk_Conn(a, in_p, i, \mathrm{IN}) \\ uact = mk_act(cs, ls, insf(RTN(a), cs)) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{p \mapsto \sigma_1(i), a \mapsto uact\}, \{rd(in_p)\})} \;\boxed{\text{rd-Pool}}$$

VDM Function:

```
67.0  rd-Pool(RTN : RTN-Types, :, :, :, :) r : 𝔹
  .1  pre let i ∈ dom  be st is-ida((i)) ∧
  .2                         RTN(i).protocol = POOL in
  .3      let a ∈ dom  in
  .4      let mk-act(cs, ls, WAIT_RD) = (a) in
  .5      let s ∈ RTN(a).sm.ss be st s.id = cs in
  .6      let op ∈ s.ops be st op.input-parameter ≠ nil in
  .7      let c ∈ dom RTN in
  .8      RTN(c) = mk-Connection(a, op.input-parameter, i, IN)
  .9  post let i ∈ dom  be st is-ida((i)) ∧
 .10                         RTN(i).protocol = POOL in
 .11      let a ∈ dom  in
 .12      let mk-act(cs, ls, WAIT_RD) = (a) in
 .13      let s ∈ RTN(a).sm.ss be st s.id = cs in
 .14      let op ∈ s.ops be st hd (cs).ops = op.id ∧
 .15                         op.input-parameter ≠ nil in
 .16      let c ∈ dom RTN be st RTN(c) = mk-Connection(a, op.input-parameter, i, IN) in
 .17      let uact = mk-act(cs, ls, insf(RTN(a), cs)) in
 .18       = †{op.input-parameter ↦ (i), a ↦ uact} ∧
 .19       = ⌢[{mk-token("rd_input_parameter")}] ;
```

**Write**

Plotkin Rule:

$a, c, i \in \mathrm{dom}\, \sigma_1$
$\sigma_1(i) = mk_IDA(vals)$
$RTN(i).protocol = \mathrm{POOL}$
$\sigma_1(a) = mk_ACT(cs, ls, \mathrm{WAIT_WE})$
$s \in RTN(a).sm.ss$
$mk_op(id, rd, we, in_p, out_p, pre, post) \in s.ops \wedge \mathrm{hd}\, \sigma_1(cs).ops = id$
$out_p \neq \mathbf{nil}$
$RTN(c) = mk_Conn(a, RTN(cs).out_p, i, \mathrm{OUT})$
$uact = mk_ACT(cs, ls, ins(RTN(cs))$

[we-Pool]

$(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{p \mapsto \sigma_1(i), a \mapsto uact\}, \{we(in_p)\})$

VDM Function:

```
68.0  we-Pool(RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let i ∈ dom  be st is-ida( (i)) ∧
 .2                       RTN (i).protocol = POOL in
 .3       let a ∈ dom  in
 .4       let mk-act(cs, ls, WAIT_WE) = (a) in
 .5       let s ∈ RTN (a).sm.ss be st s.id = cs in
 .6       let op ∈ s.ops be st op.output-result ≠ nil in
 .7       let c ∈ dom RTN in
 .8       RTN (c) = mk-Connection(a, op.output-result, i, OUT)
 .9   post let i ∈ dom  be st is-ida( (i)) ∧
 .10                      RTN (i).protocol = POOL in
 .11      let a ∈ dom  in
 .12      let mk-act(cs, ls, WAIT_WE) = (a) in
 .13      let s ∈ RTN (a).sm.ss be st s.id = cs in
 .14      let op ∈ s.ops be st op.output-result ≠ nil in
 .15      let c ∈ dom RTN be st RTN (c) = mk-Connection(a, op.output-result, i, OUT) in
 .16      let uact = mk-act(cs, ls, insf (RTN (a), cs)) in
 .17       = †{op.output-result ↦ (i), a ↦ uact} ∧
 .18       = ⁀[{mk-token("we_output_result")}] ;
```

**Channel**

**Read**

Plotkin Rule:

$$\frac{\begin{array}{l} a,c,i \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(i) = mk_IDA(vals) \\ RTN(i) = mk_IDA(id, size, \mathrm{CHANNEL}) \\ \mathrm{len}\,vals > 0 \\ \sigma_1(a) = mk_ACT(cs, ls, \mathrm{WAIT_RD}) \\ s \in RTN(a).sm.ss \\ mk_op(id, rd, we, in_p, out_p, pre, post) \in s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p \neq \mathbf{nil} \\ RTN(c) = mk_Conn(a, RTN(cs).in_p, i, \mathrm{IN}) \\ uact = mk_ACT(cs, ls, \mathrm{INDS}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{i \mapsto \mathrm{tl}\,vals, in_p \mapsto \mathrm{hd}\,vals, a \mapsto uact\}, \{rd(in_p)\}} \;\text{[rd-Chan]}$$

VDM Function:

```
69.0 rd-Chan(RTN : RTN-Types, :, :, :, :) r : B
 .1  pre let i ∈ dom  be st
 .2            is-ida( (i)) ∧
 .3            len (i).ls > 0 ∧
 .4            RTN (i).protocol = CHAN in
 .5      let a ∈ dom  in
 .6      let mk-act(cs, ls, WAIT_RD) = (a) in
 .7      let s ∈ RTN (a).sm.ss be st s.id = cs in
 .8      let op ∈ s.ops be st op.input-parameter ≠ nil in
 .9      let c ∈ dom RTN in
 .10     RTN (c) = mk-Connection(a, op.input-parameter, i, IN)
 .11 post let i ∈ dom be st is-ida( (i)) ∧
 .12                    RTN (i).protocol = CHAN in
 .13      let a ∈ dom  in
 .14      let mk-act(cs, ls, WAIT_RD) = (a) in
 .15      let s ∈ RTN (a).sm.ss be st s.id = cs in
 .16      let op ∈ s.ops be st op.input-parameter ≠ nil in
 .17      let c ∈ dom RTN be st RTN (c) = mk-Connection(a, op.input-parameter, i, IN) in
 .18      let uact = mk-act(cs, ls, insf (RTN (a), cs)) in
 .19       = †{i ↦ tl (i).ls, op.input-parameter ↦ hd (i).ls, a ↦ uact} ∧
 .20       = ⁀[{mk-token("rd_input_parameter")}] ;
```

**Write**

Plotkin Rule:

$$\frac{\begin{array}{l} a,c,i \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(i) = mk_IDA(vals) \\ RTN(i) = mk_IDA(id, size, \text{CHANNEL}) \\ size < \mathrm{len}\, vals \\ \sigma_1(a) = mk_ACT(cs, ls, \text{WAIT_WE}) \\ s \in RTN(a).sm.ss \\ mk_op(id, rd, we, in_p, out_p, pre, post) \in s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ out_p \neq \mathbf{nil} \\ RTN(c) = mk_Conn(a, RTN(cs).out_p, i, \text{OUT}) \\ uact = mk_ACT(cs, ls, \text{INDS}) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{i \mapsto \sigma(i) \frown [\sigma(out_p)], a \mapsto uact\}, \{we(out_p)\}} \;\boxed{\text{we-Chan}}$$

Note, the implicit *blocking* caused by $size < \mathrm{len}\, vals$.

VDM Function:

```
70.0  we-Chan (RTN : RTN-Types, :, :, :) r : 𝔹
  .1  pre let i ∈ dom  be st
  .2            is-ida ( (i)) ∧
  .3            RTN (i).size < len (i).ls ∧
  .4            RTN (i).protocol = CHAN in
  .5      let a ∈ dom  in
  .6      let mk-act (cs, ls, WAIT_WE) = (a) in
  .7      let s ∈ RTN (a).sm.ss be st s.id = cs in
  .8      let op ∈ s.ops be st op.output-result ≠ nil in
  .9      let c ∈ dom RTN in
  .10     RTN (c) = mk-Connection (a, op.output-result, i, OUT)
  .11 post let i ∈ dom  be st is-ida ( (i)) ∧
  .12                    RTN (i).protocol = CHAN in
  .13      let a ∈ dom  in
  .14      let mk-act (cs, ls, WAIT_WE) = (a) in
  .15      let s ∈ RTN (a).sm.ss be st s.id = cs in
  .16      let op ∈ s.ops be st op.output-result ≠ nil in
  .17      let c ∈ dom RTN be st RTN (c) = mk-Connection (a, op.output-result, i, OUT) in
  .18      let uact = mk-act (cs, ls, insf (RTN (a), cs)) in
  .19       = †{i ↦ (i).ls ⁀ [(op.output-result)], a ↦ uact} ∧
  .20       = ⁀[{mk-token ("we_output_result")}] ;
```

## C.4 Fault Semantics

What follows in this section are the Plotkin-style SOS rules for faults considered of RTNs. We first define the *fault transition* rule which defines the *fault actions* at the state-machine level.

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \text{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, mk_fault(f)) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, l) \wedge src = cs \wedge l = f \end{array}}{(RTN, \sigma_1, \pi_1 \curvearrowright [es]) \xrightarrow{f} (\sigma_1 \dagger \{a \mapsto mk_act(trg, ls, insf(trg))\}, es \cup \{f, \downarrow cs, \uparrow trg\})} \quad \boxed{\text{fault-transition}}$$

VDM Function:

71.0 $FaultTr\,(RTN : RTN\text{-}Types, :, :, :, \curvearrowright[es] :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom in

.2 let mk-$act\,(cs, ls,$ FAULT$) = (a)$ in

.3 $\exists tr \in RTN\,(a).sm.ts \cdot tr.src = cs \wedge$ is_$(tr.l, Fault)$

.4 post let $a \in$ dom be st

.5 mk-$act\,((a).cs, (a).ls,$ FAULT$) = (a)$ in

.6 let $tr \in RTN\,(a).sm.ts$ be st $tr.src = (a).cs \wedge$ is_$(tr.l, Fault)$ in

.7 let $uact =$ mk-$act\,(tr.trg, (a).ls, insf\,(RTN\,(a), tr.trg))$ in

.8 $= \dagger\{a \mapsto uact\} \wedge$

.9 $\curvearrowright[es] = \curvearrowright[es \cup \{$mk-token ("$fault$"), mk-token ("$cs$"), mk-token ("$up_tr.trg$")$\}]$ ;

## C.4.1 Late Exit Fault

Depending on the current state (dynamic/static), *cs* of an activity, the semantics for a ***late exit fault*** are as follows.

### C.4.1.1 Static State

Given the exit transition from a static state is *Time Bounded*, then the rule is as follows:

Plotkin Rule:

$$\text{[st-late-time-exit-fault]}\ \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, mk_TimeBound(l,u)) \wedge src = cs \\ \exists t_1 \in \mathrm{inds}\,\pi \cdot \uparrow cs \in_i \pi(t_1) \wedge t_1 + l \leq t_1 + u < \mathrm{len}\,\pi \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow cs \in_i \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{late_exit_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}$$

VDM Function:

```
72.0 stLateTimeExitFault(RTN : RTN-Types, :, :, :, :) r : B
 .1  pre let a ∈ dom be st (a).status = WAIT_TR in
 .2      let tr ∈ RTN(a).sm.ts be st tr.src = (a).cs ∧ is_(tr.l, TimeBound) in
 .3      ∃t1 ∈ inds ·
 .4          mk-token("up_cs") ∈ (t1) ∧
 .5          t1 + tr.l.lower ≤ t1 + tr.l.upper ∧
 .6          t1 + tr.l.upper < len ∧
 .7          (¬∃t2 ∈ inds · mk-token("dwn_cs") ∈ (t2) ∧
 .8                  t2 > t1)
 .9  post let a ∈ dom be st (a).status = WAIT_TR in
 .10      let tr ∈ RTN(a).sm.ts be st tr.src = (a).cs ∧ is_(tr.l, TimeBound) in
 .11      let t1 ∈ inds be st
 .12          mk-token("up_cs") ∈ (t1) ∧
 .13          t1 + tr.l.lower ≤ t1 + tr.l.upper ∧
 .14          t1 + tr.l.upper < len ∧
 .15          (¬∃t2 ∈ inds · mk-token("dwn_cs") ∈ (t2) ∧
 .16                  t2 > t1) in
 .17      let uact = mk-act((a).cs, (a).ls, FAULT) in
 .18      = †{a ↦ uact} ∧
 .19      = ⁀[{mk-token("late_exit_fault")}] ;
```

Given the exit transition from a static state is *Event Bounded*, then the rule is as follows:

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, mk_Event(e)) \wedge src = cs \\ \exists t_1, t_1' \in \mathrm{inds}\,\pi \cdot t_1 < t_1' \wedge \uparrow cs \in_i \pi(t_1) \wedge e \in \pi(t_1') \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow cs \in_i \pi(t_2) \wedge t_2 \geq t_1' \\ uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_exit_fault\}]) \xrightarrow{s} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \boxed{\text{st-late-event-exit-fault}}$$

VDM Function:

```
73.0  stLateEventExitFault (RTN : RTN-Types, :, :, :) r : B
 .1   pre let a ∈ dom be st (a).status = WAIT_TR in
 .2       let tr ∈ RTN (a).sm.ts be st tr.src = (a).cs ∧ is_(tr.l, Event) in
 .3       ∃t1, t2 ∈ inds ·
 .4            t1 < t2 ∧ mk-token ("up_cs") ∈ (t1) ∧
 .5            tr.l ∈ (t2) ∧
 .6            (¬∃t3 ∈ inds · mk-token ("dwn_cs") ∈ (t3) ∧
 .7                          t3 > t2)
 .8   post let a ∈ dom be st (a).status = WAIT_TR in
 .9       let tr ∈ RTN (a).sm.ts be st tr.src = (a).cs ∧ is_(tr.l, Event) in
 .10      let t1 ∈ inds in
 .11      let t2 ∈ inds be st
 .12               t1 < t2 ∧ mk-token ("up_cs") ∈ (t1) ∧
 .13               tr.l ∈ (t2) ∧
 .14               (¬∃t3 ∈ inds · mk-token ("dwn_cs") ∈ (t3) ∧
 .15                             t3 > t2) in
 .16      let uact = mk-act ((a).cs, (a).ls, FAULT) in
 .17        = †{a ↦ uact} ∧
 .18        = ↷[{mk-token ("late_exit_fault")}] ;
```

#### C.4.1.2 Dynamic State

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ mk_Dynamic_State(id, _, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathrm{inds}\,\pi_1 \cdot \uparrow cs \in \pi(t_1) \wedge t_1 + bcet \leq t_1 + wcet < \mathrm{len}\,\pi_1 \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi_1 \cdot \downarrow cs \in \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\text{LATE_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{late_exit_fault\}]) \xrightarrow{f} (\sigma_2 . es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \; \text{[dy-late-exit-fault]}$$

VDM Function:

```
74.0  dyLateExitFault (RTN : RTN-Types, :, :, :, :) r : 𝔹
 .1   pre let a ∈ dom  be st  (a).status = INDS in
 .2       let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
 .3               id = (a).cs in
 .4       ∃t1 ∈ inds  ·
 .5            mk-token ("up_cs") ∈ (t1) ∧
 .6            t1 + bcet ≤ t1 + wcet ∧
 .7            t1 + wcet < len  ∧
 .8            (¬∃t2 ∈ inds  · mk-token ("dwn_cs") ∈ (t2) ∧
 .9                          t2 > t1)
 .10  post let a ∈ dom  be st  (a).status = INDS in
 .11      let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
 .12              id = (a).cs in
 .13      let t1 ∈ inds  be st
 .14              mk-token ("up_cs") ∈ (t1) ∧
 .15              t1 + bcet ≤ t1 + wcet ∧
 .16              t1 + wcet < len  in
 .17      let t2 ∈ inds  be st mk-token ("dwn_cs") ∉ (t2) ∧
 .18                          t2 > t1 in
 .19      let uact = mk-act ( (a).cs, (a).ls, FAULT) in
 .20       = †{a ↦ uact} ∧
 .21       = ⁀[{mk-token ("late_exit_fault")}] ;
```

## C.4.2 Early Exit Fault

Depending on the current state (dynamic/static), *cs* of an activity, the semantics for a *early exit fault* are as follows.

### C.4.2.1 Static State

Given the exit transition from a static state is *Time Bounded*, then the rule is as follows:

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{WAIT_TR}) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, mk_TimeBound(l,u)) \wedge src = cs \\ \exists t_1 \in \mathrm{inds}\,\pi \cdot \uparrow cs \in_i \pi(t_1) \wedge \mathrm{len}\,\pi < t_1 + l \leq t_1 + u \wedge \\ \qquad \exists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow cs \in_i \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{EARLY_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{early_exit_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \boxed{\text{st-early-time-exit-fault}}$$

VDM Function:

```
75.0  stEarlyTimeExitFault (RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom  be st  (a).status = WAIT_TR in
 .2       let tr ∈ RTN (a).sm.ts be st  tr.src = (a).cs ∧ is_(tr.l, TimeBound) in
 .3       ∃t1 ∈ inds  ·
 .4            mk-token ("up_cs") ∈ (t1) ∧
 .5            len  < t1 + tr.l.lower ∧
 .6            t1 + tr.l.lower < t1 + tr.l.upper ∧
 .7            (∃t2 ∈ inds  · mk-token ("dwn_cs") ∈ (t2) ∧
 .8                        t2 > t1)
 .9   post let a ∈ dom  be st  (a).status = WAIT_TR in
 .10       let tr ∈ RTN (a).sm.ts be st  tr.src = (a).cs ∧ is_(tr.l, TimeBound) in
 .11       let t1 ∈ inds  be st
 .12             mk-token ("up_cs") ∈ (t1) ∧
 .13             len  < t1 + tr.l.lower ∧
 .14             t1 + tr.l.lower < t1 + tr.l.upper ∧
 .15             (∃t2 ∈ inds  · mk-token ("dwn_cs") ∈ (t2) ∧
 .16                         t2 > t1) in
 .17       let uact = mk-act ( (a).cs, (a).ls, FAULT) in
 .18        = †{a ↦ uact} ∧
 .19        = ⁀[{mk-token ("early_exit_fault")}] ;
```

Given the exit transition from a static state is *Event Bounded*, then the rule is as follows:

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{WAIT_TR}) \\ \exists t \in RTN(a).sm.ts \cdot mk_Transition(src, trg, mk_Event(e)) \wedge src = cs \\ \exists t_1, t_1' \in \mathrm{inds}\,\pi \cdot t_1 < t_1' \wedge \uparrow cs \in_i \pi(t_1) \wedge e \in \pi(t_1') \wedge \\ \qquad\qquad \exists t_2 \in \mathrm{inds}\,\pi \cdot \downarrow cs \in_i \pi(t_2) \wedge t_2 < t_1' \\ uact = mk_act(cs, ls, mk_fault(\text{EARLY_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{early_exit_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \boxed{\text{st-early-event-exit-fault}}$$

VDM Function:

76.0 $stEarlyEventExitFault(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st $(a).status =$ WAIT_TR in

.2 let $tr \in RTN\,(a).sm.ts$ be st $tr.src = (a).cs \wedge$ is_$(tr.l, Event)$ in

.3 $\exists t1, t2 \in$ inds $\cdot$

.4 $t1 < t2 \wedge$ mk-token ("$up_cs$") $\in (t1) \wedge$

.5 $tr.l \in (t2) \wedge$

.6 $(\exists t3 \in$ inds $\cdot$ mk-token ("$dwn_cs$") $\in (t3) \wedge$

.7 $t3 < t2)$

.8 post let $a \in$ dom be st $(a).status =$ WAIT_TR in

.9 let $tr \in RTN\,(a).sm.ts$ be st $tr.src = (a).cs \wedge$ is_$(tr.l, Event)$ in

.10 let $t1 \in$ inds in

.11 let $t2 \in$ inds be st

.12 $t1 < t2 \wedge$ mk-token ("$up_cs$") $\in (t1) \wedge$

.13 $tr.l \in (t2) \wedge$

.14 $(\exists t3 \in$ inds $\cdot$ mk-token ("$dwn_cs$") $\in (t3) \wedge$

.15 $t3 < t2)$ in

.16 let $uact =$ mk-$act((a).cs, (a).ls,$ FAULT$)$ in

.17 $= \dagger\{a \mapsto uact\} \wedge$

.18 $= \curvearrowright [\{$mk-token ("$early_exit_fault$")$\}]$ ;

#### C.4.2.2 Dynamic State

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathsf{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ mk_Dynamic_State(id, _, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ \exists t_1 \in \mathsf{inds}\,\pi_1 \cdot \uparrow cs \in \pi(t_1) \wedge t_1 + bcet < \mathsf{len}\,\pi_1 \leq t_1 + wcet \wedge \\ \qquad \exists t_2 \in \mathsf{inds}\,\pi_1 \cdot \downarrow cs \in \pi(t_2) \wedge t_2 < \mathsf{len}\,\pi_1 \\ uact = mk_act(cs, ls, mk_fault(\text{EARLY_EXIT})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{early_exit_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \boxed{\text{dy-early-exit-fault}}$$

VDM Function:

```
77.0  dyEarlyExitFault (RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom  be st  (a).status = INDS in
 .2       let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
 .3              id = (a).cs in
 .4       ∃t1 ∈ inds  ·
 .5            mk-token ("up_cs") ∈ (t1) ∧
 .6            t1 + bcet ≤ t1 + wcet ∧
 .7            t1 + wcet < len ∧
 .8            (∃t2 ∈ inds  · mk-token ("dwn_cs") ∈ (t2) ∧
 .9                        t2 < len )
 .10  post let a ∈ dom  be st  (a).status = INDS in
 .11      let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
 .12             id = (a).cs in
 .13      let t1 ∈ inds  be st
 .14             mk-token ("up_cs") ∈ (t1) ∧
 .15             t1 + bcet ≤ t1 + wcet ∧
 .16             t1 + wcet < len  in
 .17      let t2 ∈ inds  be st mk-token ("dwn_cs") ∈ (t2) ∧
 .18                       t2 < len  in
 .19      let uact = mk-act ( (a).cs, (a).ls, FAULT) in
 .20       = †{a ↦ uact} ∧
 .21       = ⁀[{mk-token ("early_exit_fault")}] ;
```

### C.4.3 Read Faults

**Late Read:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge in_p \neq \mathbf{nil} \\ \exists t_1 \in \mathrm{inds}\,\pi_1 \cdot \uparrow\!cs \in \pi(t_1) \wedge t_1 + bcet < \mathrm{len}\,\pi_1 \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi_1 \cdot rds(in_p) \in \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_READ})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_read_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \text{dy-late-read-fault}$$

VDM Function:

78.0 $lateReadFault(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st $(a).status = \mathrm{INDS}$ in

.2 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss$ be st

.3 $id = (a).cs$ in

.4 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.5 $((a).cs).ops \neq [] \wedge$

.6 hd $((a).cs).ops = id \wedge$

.7 $in\text{-}p \neq$ nil in

.8 $\exists t1 \in$ inds $\cdot$

.9 mk-token ("$up_cs$") $\in (t1) \wedge$

.10 $t1 + bcet <$ len $\wedge$

.11 $(\neg \exists t2 \in$ inds $\cdot$ mk-token ("$rd(in_p)$") $\in (t2) \wedge$

.12 $t2 > t1)$

.13 post let $a \in$ dom be st $(a).status =$ INDS in

.14 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss$ be st

.15 $id = (a).cs$ in

.16 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.17 $((a).cs).ops \neq [] \wedge$

.18 hd $((a).cs).ops = id \wedge$

.19 $in\text{-}p \neq$ nil in

.20 let $t1 \in$ inds be st

.21 mk-token ("$up_cs$") $\in (t1) \wedge$

.22 $t1 + bcet <$ len $\wedge$

.23 $(\neg \exists t2 \in$ inds $\cdot$ mk-token ("$rd(in_p)$") $\in (t2) \wedge$

.24 $t2 > t1)$ in

.25 let $uact =$ mk-$act((a).cs, (a).ls,$ FAULT) in

.26 $= \dagger\{a \mapsto uact\} \wedge$

.27 $= \curvearrowright[\{$mk-token ("$late_read_fault$")$\}]$ ;

**Early Read:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge in_p \neq \mathbf{nil} \\ \exists t_1 \in \mathrm{inds}\,\pi_1 \cdot \uparrow cs \in \pi(t_1) \wedge \\ \qquad \exists t_2 \in \mathrm{inds}\,\pi_1 \cdot rds(in_p) \in \pi(t_2) \wedge t_2 < t_1 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{EARLY_READ})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{early_read_fault\}]) \xrightarrow{s} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \text{dy-early-read-fault}$$

VDM Function:

79.0 $earlyReadFault(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st $(a).status =$ INDS in

.2 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss$ be st

.3 $id = (a).cs$ in

.4 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.5 $((a).cs).ops \neq [] \wedge$

.6 hd $((a).cs).ops = id \wedge$

.7 $in\text{-}p \neq$ nil in

.8 $\exists t1 \in$ inds $\cdot$ mk-token ("$up_cs$") $\in (t1) \wedge$

.9 $(\exists t2 \in$ inds $\cdot$ mk-token ("$rd(in_p)$") $\in (t2) \wedge$

.10 $t2 < t1)$

.11 post let $a \in$ dom be st $(a).status =$ INDS in

.12 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss$ be st

.13 $id = (a).cs$ in

.14 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.15 $((a).cs).ops \neq [] \wedge$

.16 hd $((a).cs).ops = id \wedge$

.17 $in\text{-}p \neq$ nil in

.18 let $t1 \in$ inds be st mk-token ("$up_cs$") $\in (t1) \wedge$

.19 $(\exists t2 \in$ inds $\cdot$ mk-token ("$rd(in_p)$") $\in (t2) \wedge$

.20 $t2 < t1)$ in

.21 let $uact =$ mk-$act((a).cs, (a).ls,$ FAULT$)$ in

.22 $= \dagger\{a \mapsto uact\} \wedge$

.23 $= \frown[\{$mk-token ("$early_read_fault$")$\}]$ ;

**Value Read:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p \neq \mathbf{nil}, out_p = \mathbf{nil} \\ (in_p, RTN, \sigma_1) \xrightarrow{rd} v \\ ls' = ls \dagger \{in_p \mapsto v\} \\ (pre, ls) \xrightarrow{e} \mathbf{false} \\ uact = mk_act(cs, ls', mk_fault(\mathrm{VALUE_READ})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact, cs \mapsto \mathrm{tl}\,\sigma_1(cs).ops\}, \pi_1 \curvearrowright [\{rd(in_p), value_read_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \text{dy-value-read-fault}$$

VDM Function:

80.0 $valueReadFault(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$

.1 pre let $a \in$ dom be st is_$((a), act)$ in

.2 let mk-$act(cs, ls, \mathrm{INDS}) = (a)$ in

.3 let mk-$Operation(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN(a).ops$ be st

.4 $(cs).ops \neq [] \wedge$

.5 hd $(cs).ops = id \wedge$

.6 $in\text{-}p \neq$ nil $\wedge$

.7 $out\text{-}p =$ nil in

.8 let $inp \in RTN(a).input\text{-}ports$ be st

.9 $inp.id = in\text{-}p$ in

.10 let $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp, )\}$ in

.11 $\neg expr(preC, ls')$

.12 post let $a \in$ dom be st

.13 $((a).cs).ops \neq []$ in

.14 let $op \in RTN(a).ops$ be st

.15 hd $((a).cs).ops = op.id \wedge$

.16 $op.input\text{-}parameter \neq$ nil in

.17 let $inp \in RTN(a).input\text{-}ports$ be st

.18 $inp.id = op.input\text{-}parameter$ in

.19 let $ls' : Id \xrightarrow{m} Var$ be st

.20 $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp, )\} \wedge$

.21 $\neg expr(op.preC, (a).ls')$ in

.22 let $uact =$ mk-$act(cs, ls', \mathrm{FAULT})$ in

.23 $= \dagger\{cs \mapsto$ tl $(cs).ops, a \mapsto uact\} \wedge$

.24 $= \curvearrowright[\{$mk-token("$rd(in_p)$"), mk-token("$value_read_fault$")$\}]$ ;

**Omit Read:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\, \sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge in_p \neq \mathbf{nil} \\ \exists t_1, t_2 \in \mathrm{inds}\, \pi_1 \cdot \uparrow cs \in_i \pi(t_1) \wedge \downarrow cs \in_i \pi(t_2) \wedge \\ \qquad \nexists t_2' \in \mathrm{inds}\, \pi_1 \cdot rds(in_p) \in \pi(t_2') \wedge t_1 < t_2' \leq t_2 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{OMIT_READ})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{omit_read_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \text{dy-omit-read-fault}$$

VDM Function:

```
81.0  OmitReadFault(RTN : RTN-Types, :, :, :, :) r : B
 .1   pre let a ∈ dom be st (a).status = INDS in
 .2       let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
 .3             id = (a).cs in
 .4       let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
 .5             ((a).cs).ops ≠ [] ∧
 .6             hd ((a).cs).ops = id ∧
 .7             in-p ≠ nil in
 .8       ∃t1, t2 ∈ inds · mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
 .9                 (¬∃t3 ∈ inds ·
 .10                      mk-token("rd(in_p)") ∈ (t2) ∧
 .11                      t1 < t3 ∧
 .12                      t3 ≤ t2)
 .13  post let a ∈ dom be st (a).status = INDS in
 .14      let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
 .15            id = (a).cs in
 .16      let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
 .17            ((a).cs).ops ≠ [] ∧
 .18            hd ((a).cs).ops = id ∧
 .19            in-p ≠ nil in
 .20      let t1 ∈ inds in
 .21      let t2 ∈ inds be st mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
 .22                (¬∃t3 ∈ inds ·
 .23                     mk-token("rd(in_p)") ∈ (t2) ∧
 .24                     t1 < t3 ∧
 .25                     t3 ≤ t2) in
 .26      let uact = mk-act((a).cs, (a).ls, FAULT) in
 .27       = †{a ↦ uact} ∧
 .28       = ⁀[{mk-token("omit_read_fault")}] ;
```

### C.4.4 Write Faults

**Late Write:**

Plotkin Rule:

$$
\text{[dy-late-write-fault]}\quad
\frac{
\begin{array}{l}
a \in \mathrm{dom}\,\sigma_1 \\
\sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\
mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\
mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge out_p \neq \mathbf{nil} \\
\exists t_1, t_2 \in \mathrm{inds}\,\pi_1 \cdot \uparrow\!cs \in_i \pi(t_1) \wedge \downarrow\!cs \in_i \pi(t_1) \wedge \\
\qquad\qquad \nexists t_2' \in \mathrm{inds}\,\pi_1 \cdot wds(out_p) \in \pi(t_2') \wedge t_2' > t_2 \\
uact = mk_act(cs, ls, mk_fault(\mathrm{LATE_WRITE})) \\
(RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{late_write_fault\}]) \xrightarrow{f} (\sigma_2, es)
\end{array}
}{
(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)
}
$$

VDM Function:

```
82.0  lateWriteFault (RTN : RTN-Types, :, :, :, :) r : 𝔹
  .1  pre let a ∈ dom be st (a).status = INDS in
  .2      let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
  .3              id = (a).cs in
  .4      let mk-Operation (id, read, we, in-p, out-p, preC, postC) ∈ RTN (a).ops be st
  .5              ((a).cs).ops ≠ [] ∧
  .6              hd ((a).cs).ops = id ∧
  .7              out-p ≠ nil in
  .8      ∃ t1, t2 ∈ inds · mk-token ("up_cs") ∈ (t1) ∧ mk-token ("dwn_cs") ∈ (t2) ∧
  .9                  (¬∃ t3 ∈ inds · mk-token ("we(out_p)") ∈ (t3) ∧
 .10                          t3 > t2)
 .11  post let a ∈ dom be st (a).status = INDS in
 .12      let mk-Dynamic-State (id, ops, mk-Bounds (bcet, bcrt, wcet)) ∈ RTN (a).sm.ss be st
 .13              id = (a).cs in
 .14      let mk-Operation (id, read, we, in-p, out-p, preC, postC) ∈ RTN (a).ops be st
 .15              ((a).cs).ops ≠ [] ∧
 .16              hd ((a).cs).ops = id ∧
 .17              out-p ≠ nil in
 .18      let t1 ∈ inds in
 .19      let t2 ∈ inds be st mk-token ("up_cs") ∈ (t1) ∧ mk-token ("dwn_cs") ∈ (t2) ∧
 .20                  (¬∃ t3 ∈ inds · mk-token ("we(out_p)") ∈ (t3) ∧
 .21                          t3 > t2) in
 .22      let uact = mk-act ((a).cs, (a).ls, FAULT) in
 .23      = †{a ↦ uact} ∧
 .24      = ⌒[{mk-token ("late_write_fault")}] ;
```

**Early Write:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\ mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge out_p \neq \mathbf{nil} \\ \exists t_1, t_2 \in \mathrm{inds}\,\pi_1 \cdot \uparrow cs \in_i \pi(t_1) \wedge \downarrow cs \in_i \pi(t_2) \wedge \\ \qquad \exists t_2' \in \mathrm{inds}\,\pi_1 \cdot wds(out_p) \in \pi(t_2') \wedge t_2' < t_2 \\ uact = mk_act(cs, ls, mk_fault(\mathrm{EARLY_WRITE})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{early_write_fault\}]) \xrightarrow{s} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \boxed{\text{dy-early-write-fault}}$$

VDM Function:

```
83.0 earlyWriteFault(RTN : RTN-Types, :, :, :, :) r : B
 .1  pre let a ∈ dom be st (a).status = INDS in
 .2      let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
 .3              id = (a).cs in
 .4      let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
 .5              ((a).cs).ops ≠ [] ∧
 .6              hd ((a).cs).ops = id ∧
 .7              out-p ≠ nil in
 .8      ∃t1, t2 ∈ inds · mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
 .9                  (∃t3 ∈ inds · mk-token("we(out_p)") ∈ (t3) ∧
 .10                             t3 < t2)
 .11 post let a ∈ dom be st (a).status = INDS in
 .12      let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
 .13              id = (a).cs in
 .14      let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
 .15              ((a).cs).ops ≠ [] ∧
 .16              hd ((a).cs).ops = id ∧
 .17              out-p ≠ nil in
 .18      let t1 ∈ inds in
 .19      let t2 ∈ inds be st mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
 .20                  (∃t3 ∈ inds · mk-token("we(out_p)") ∈ (t3) ∧
 .21                             t3 < t2) in
 .22      let uact = mk-act((a).cs, (a).ls, FAULT) in
 .23       = †{a ↦ uact} ∧
 .24       = ⁀[{mk-token("early_write_fault")}] ;
```

**Value Write:**

Plotkin Rule:

$$\frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\ \sigma_1(cs).ops \neq [] \\ s \in RTN(a).sm.ss \\ mk_Operation(id, rd, we, in_p, out_p, pre, post) \in \mathrm{elems}\, s.ops \wedge \mathrm{hd}\,\sigma_1(cs).ops = id \\ in_p = \mathbf{nil}, out_p \neq \mathbf{nil} \\ (out_p, v', ls) \xrightarrow{we} ls' \\ (post, ls') \xrightarrow{e} \mathbf{false} \\ uact = mk_act(cs, ls', mk_fault(\mathrm{VALUE_WRITE})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact, cs \mapsto \mathrm{tl}\,\sigma_1(cs).ops\}, \pi_1 \frown [\{we(out_p), value_write_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)} \quad \text{dy-value-write-fault}$$

VDM Function:

84.0 $valueWriteFault\,(RTN : RTN\text{-}Types, :, :, :)\; r : \mathbb{B}$

.1 pre let $a \in \mathrm{dom}$ be st is_( $(a), act)$ in

.2 let mk-$act\,(cs, ls, \mathrm{INDS}) = (a)$ in

.3 let mk-$Operation\,(id, read, we, in\text{-}p, out\text{-}p, preC, postC) \in RTN\,(a).ops$ be st

.4 $(cs).ops \neq [] \wedge$

.5 hd $(cs).ops = id \wedge$

.6 $in\text{-}p \neq$ nil $\wedge$

.7 $out\text{-}p =$ nil in

.8 let $inp \in RTN\,(a).input\text{-}ports$ be st

.9 $inp.id = in\text{-}p$ in

.10 let $ls' = (a).ls \dagger \{inp.id \mapsto rdport\,(inp, )\}$ in

.11 $expr\,(preC, ls') \wedge$

.12 let $outp \in RTN\,(a).output\text{-}ports$ be st

.13 $outp.id = out\text{-}p$ in

.14 let $ls'' : Id \xrightarrow{m} Var$ in

.15 $\exists v : Var \cdot$

.16 $ls'' = (a).ls \dagger ls' \dagger \{outp.id \mapsto weport\,(outp, v, )\} \wedge$

.17 $\neg expr\,(postC, ls'')$

.18 post let $a \in$ dom be st
.19 $((a).cs).ops \neq []$ in
.20 let $op \in RTN(a).ops$ be st
.21 hd $((a).cs).ops = op.id \wedge$
.22 $op.input\text{-}parameter \neq$ nil in
.23 let $inp \in RTN(a).input\text{-}ports$ be st
.24 $inp.id = op.input\text{-}parameter$ in
.25 let $ls' : Id \xrightarrow{m} Var$ be st
.26 $ls' = (a).ls \dagger \{inp.id \mapsto rdport(inp,)\} \wedge$
.27 $expr(op.preC, (a).ls')$ in
.28 let $outp \in RTN(a).output\text{-}ports$ be st
.29 $outp.id = op.output\text{-}result$ in
.30 let $ls'' : Id \xrightarrow{m} Var$ be st
.31 $\exists v : Var \cdot$
.32 $ls'' = (a).ls \dagger ls' \dagger \{outp.id \mapsto weport(outp, v,)\} \wedge$
.33 $\neg expr(op.postC, ls'')$ in
.34 let $uact =$ mk-$act(cs, ls', \text{FAULT})$ in
.35 $= \dagger\{cs \mapsto$ tl $(cs).ops, a \mapsto uact\} \wedge$
.36 $= \curvearrowright[\{$mk-token ("$rd(in_p)$"), mk-token ("$value_write_fault$")$\}]$ ;

**Omit Write:**

Plotkin Rule:

$$
\begin{array}{l}
a \in \mathrm{dom}\,\sigma_1 \\
\sigma_1(a) = mk_act(cs, ls, \mathrm{INDS}) \\
mk_Dynamic_State(id, ops, mk_Bounds(bcet, bcrt, wcet)) \in RTN(a).sm.ss \wedge id = cs \\
mk_Operation(_,_,_, in_p, out_p, _,_) \in ops \wedge out_p \neq \mathbf{nil} \\
\exists t_1, t_2 \in \mathrm{inds}\,\pi_1 \cdot \uparrow cs \in_i \pi(t_1) \wedge \downarrow cs \in_i \pi(t_2) \wedge \\
\qquad\qquad \nexists t_2' \in \mathrm{inds}\,\pi_1 \cdot wds(out_p) \in \pi(t_2) \wedge t_1 < t_2' \leq t_2 \\
uact = mk_act(cs, ls, mk_fault(\mathrm{OMIT_WRITE})) \\
(RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \curvearrowright [\{omit_write_fault\}]) \xrightarrow{f} (\sigma_2, es) \\
\hline
(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)
\end{array}
\quad \boxed{\text{dy-omit-write-fault}}
$$

VDM Function:

```
85.0  OmitWriteFault(RTN : RTN-Types, :, :, :, :) r : 𝔹
  .1  pre let a ∈ dom  be st (a).status = INDS in
  .2      let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
  .3              id = (a).cs in
  .4      let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
  .5              ((a).cs).ops ≠ [] ∧
  .6              hd ((a).cs).ops = id ∧
  .7              out-p ≠ nil in
  .8      ∃t1, t2 ∈ inds · mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
  .9                      (¬∃t3 ∈ inds ·
  .10                          mk-token("we(out_p)") ∈ (t2) ∧
  .11                          t1 < t3 ∧
  .12                          t3 ≤ t2)
  .13  post let a ∈ dom  be st (a).status = INDS in
  .14      let mk-Dynamic-State(id, ops, mk-Bounds(bcet, bcrt, wcet)) ∈ RTN(a).sm.ss be st
  .15              id = (a).cs in
  .16      let mk-Operation(id, read, we, in-p, out-p, preC, postC) ∈ RTN(a).ops be st
  .17              ((a).cs).ops ≠ [] ∧
  .18              hd ((a).cs).ops = id ∧
  .19              out-p ≠ nil in
  .20      let t1 ∈ inds in
  .21      let t2 ∈ inds be st mk-token("up_cs") ∈ (t1) ∧ mk-token("dwn_cs") ∈ (t2) ∧
  .22                      (¬∃t3 ∈ inds ·
  .23                          mk-token("we(out_p)") ∈ (t2) ∧
  .24                          t1 < t3 ∧
  .25                          t3 ≤ t2) in
  .26      let uact = mk-act((a).cs, (a).ls, FAULT) in
  .27      = †{a ↦ uact} ∧
  .28      = ⌒[{mk-token("omit_write_fault")}] ;
```

### C.4.5 Crash Faults

Plotkin Rule:

$$\text{[dy-crash-fault]}\ \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ \sigma_1(a) = mk_act(cs, ls, \text{INDS}) \\ \exists t_1 \in \mathrm{inds}\,\pi_1 \cdot \nmid cs \in_i \pi(t_1) \wedge \\ \qquad \nexists t_2 \in \mathrm{inds}\,\pi_1 \cdot \nmid cs \in_i \pi(t_2) \wedge t_2 > t_1 \\ uact = mk_act(cs, ls, mk_fault(\text{CRASH})) \\ (RTN, \sigma_1 \dagger \{a \mapsto uact\}, \pi_1 \frown [\{crash_fault\}]) \xrightarrow{f} (\sigma_2, es) \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_2, es)}$$

VDM Function:

86.0 $crashFault(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$
.1 pre let $a \in$ dom be st $(a).status = \text{INDS}$ in
.2 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN\,(a).sm.ss$ be st
.3 $id = (a).cs$ in
.4 $\exists t1 \in$ inds $\cdot$ mk-token $("up_cs") \in (t1) \wedge$
.5 $(\neg \exists t2 \in$ inds $\cdot$ mk-token $("dwn_cs") \in (t2) \wedge$
.6 $t2 > t1)$
.7 post let $a \in$ dom be st $(a).status = \text{INDS}$ in
.8 let mk-$Dynamic\text{-}State(id, ops,$ mk-$Bounds(bcet, bcrt, wcet)) \in RTN\,(a).sm.ss$ be st
.9 $id = (a).cs$ in
.10 let $t1 \in$ inds be st mk-token $("up_cs") \in (t1)$ in
.11 let $t2 \in$ inds be st mk-token $("dwn_cs") \notin (t2) \wedge$
.12 $t2 > t1$ in
.13 let $uact =$ mk-$act((a).cs, (a).ls, \text{FAULT})$ in
.14 $= \dagger\{a \mapsto uact\} \wedge$
.15 $= \frown[\{$mk-token $("late_exit_fault")\}]$

## C.5 Implementation Considerations

"Concurrency is present in labelled transition systems (LTSs) only as an interpretation of non-determinism." [Pay93]

We require a transition system which distinguishes between the intermediate states -configurations- during an evaluation of an RTN. This requires a fine-grained semantics which is commonly referred to as a *small step* semantic, which requires judgements for single steps in the evaluation. Conversely, a *large step* semantics is one which evaluates a construct (in our case an RTN) and returns the final state and trace. The semantic rules we have derived and described previously are sufficiently fine-grained and constitute a small step semantic.

Addressing the issue of concurrency, having a static specification ($rtn$), the interpretation of concurrency is clear. Given a configuration $(RTN, \sigma_1, \pi_1)$, any rules whoms premises are satisfied should fire concurrently and introduce concurrency into our semantics.

## C.5.1 An Executable Specification (for Animation)

The implication to implement $\xrightarrow{rtn}$ is to consider ramifying the results of the (powerset) relation $\xrightarrow{s}$. Considering an implementation of $\xrightarrow{s}$ we explore an executable specification of $\xrightarrow{s}$ in VDM-SL. This exploration unveils the pitfalls of giving an executable specification [Fuc92, HJ89] of a predicate, requiring we ramify the resulting *set of configurations*. Three possible implementations are considered and contrasted and a final decision made. We motivate the benefits of such a model and outline its contribution to first the verification of the SOS rules, and secondly, its contribution to animating the semantics in an interpreter-like fashion.

To design and specify the semantics of a programming /specification language involves a number of clerical task, such as rendering, type-checking and animation. Tools help elevate some of this burden as well as impose confidence in the validity of the specifications. However, the tools currently available [tea94, vDHK96] exhibit considerable variation in their sophistication.

We begin by exploring the benefits of using the VDMToolsand draw observations from our own work which then motivates a switch to using existing tool support - LETOS [Har97]. Our key observation is supported by the work that surrounds LETOS, for which we give an introduction. The issues with regards animation are a running theme throughout the discussion.

### C.5.1.1 Modelling non-determinism

Adopting the Plotkin-style rule format, the issue of concurrency and non-determinism is factored out to a "meta-level". It is this meta-level that is of interest here. Although the notation used separates out the issue of the order in which to apply the rules from that of the relation between states, the complication of writing a function which directly defines the set of all possible states remains [Jon03].

Whatever the origin of the non-determinism, it is clear that

$$exec : RTN \times \Sigma \times \Pi \rightarrow \Sigma \times \Pi$$

does not capture the semantics intent. A move to producing a set of states, as in

$$exec : RTN \times \Sigma \times \Pi \rightarrow \mathcal{P}(\Sigma \times \Pi)$$

is not convenient because of the need to ramify the combinations. The key advantage of a rule format such as:

$$\textsf{term}\quad \frac{\begin{array}{l} a \in \mathrm{dom}\,\sigma_1 \\ is_(\sigma_1(a), act) \\ \nexists tr \in RTN(a).sm.ts \cdot tr.src = \sigma_1(a).cs \end{array}}{(RTN, \sigma_1, \pi_1) \xrightarrow{s} (\sigma_1 \dagger \{a \mapsto mk_act(\sigma_1(a).cs, \sigma_1(a).ls.\mathrm{TERM})\}, \{\})}$$

is that it provides a natural way of expressing the relation

$$\xrightarrow{s} : \mathcal{P}((RTN \times \Sigma \times \Pi) \times \Sigma \times \Pi)$$

Given this relation, we wish to specify the "meta-level" with a predicate such as:

87.0 $strans(RTN : RTN\text{-}Types, :, :, :, :)\ r : \mathbb{B}$
.1 post if $all\text{-}term(RTN, , )$
.2 then $= \wedge =$
.3 else let mk- $(,) = ex\text{-}strans(RTN, , )$ in
.4 $\exists :, : \cdot strans(RTN, , , , ) \wedge = \wedge =\ ;$

which specifies entirely the non-determinism in our model expressed in the $\xrightarrow{s}$ relation.

We are now interested in modelling this non-determinism in an explicit executable VDM-SL model, and the next section articulates our ideas on how to animate our semantics.

#### C.5.1.2 Implementation in the VDMTools

From the set of SOS rules as described in Section 5.2.1, we represent each rule as an explicit VDM-SL function. However, VDM-SL -as implemented in the VDMTools- prohibits specifying a mechanism that chooses the next rule to *fire* in a configurations $(RTN, \sigma_1, \pi_1)$ non-deterministically. We address this issue and those described previously. We assume a set of function identifiers, which are the identifiers to the explicit specifications of our SOS rules, form a set (*world*) which represents the total options for making a transition in our semantic model. From this set, we can find the subset of *enabled* functions whose premises are true in a given configuration. From this set, *enabled* we can specify non-deterministically which rule to fire. However, the VDM-SL implementation will always choose the same element from a set.

The remaining issue is given the set of *enabled* functions, should we evaluate each element i) in turn, ii) in parallel, or iii) recursively. We now specify an implementation of this "meta-level" as *exec*1(), *exec*2() and *exec*3() respectively and discuss the merits for each.

**Three possible implementations of the "meta-level"**

First, that the execution is based on recursion where one of the enabled functions from $(\sigma_1, \pi_1)$ is evaluated, then recurse until all activities in the network have terminated.

```
88.0  exec: RTN-Types × × → ×
  .1  exec(RTN,,) △
  .2    if ¬all-term(RTN,,)
  .3    then let f ∈ select(world, {}, RTN,,) in
  .4        cases f :
  .5          F1 → let mk-(,) = f1 (RTN,,) in
  .6                exec(RTN,,),
  .7          F2 → let mk-(,) = f2 (RTN,,) in
  .8                exec(RTN,,)
  .9        end
  .10   else mk-(,);
```

- This implementation returns **one of** the *possible* traces.

Secondly, that **all** enabled functions in $(\sigma_1, \pi_1)$ are executed in **parallel** and the resulting configurations ramified. This relies on the fact the rules are non-interfering[3].

```
89.0  exec2: RTN-Types × × → ×
  .1  exec2(RTN,,) △
  .2    if ¬all-term(RTN,,)
  .3    then let enabled = select(world, {}, RTN,,) in
  .4        let mk-(,) = merge-all({cases f :
  .5                                  F1 → f1 (RTN,,),
  .6                                  F2 → f2 (RTN,,)
  .7                                end |
  .8                                  f ∈ enabled}) in
  .9        exec2(RTN,,)
  .10   else mk-(,);
```

- This implementation returns **the ramified** set of all the *possible* traces.

Thirdly, that **all** enabled functions in $(\sigma_1, \pi_1)$ are executed **recursively**, in that each function is **evaluated** on the result of the previous recursive step, until all activities are terminated.

```
90.0  exec3: RTN-Types × × → ×
  .1  exec3(RTN,,) △
  .2    if ¬all-term(RTN,,)
  .3    then let enabled = select(world, {}, RTN,,) in
  .4        let mk-(,) = recurse(enabled, RTN,,) in
  .5        exec3(RTN,,)
  .6    else mk-(,);
```

[3]Each activity is essentially a sequential loop (with possible non-deterministic paths) only one rule for one activity will be able to 'fire', therefore we can be sure another rule will not interfere

- This implementation returns the **greedy** semantics, such that all the *possible* execution paths are evaluated and ramified.

### Discussion

Each proposal proposal has its merits, but only one realises an adequate specification of the concurrency & non-determinism required - that is *exec3()*. However, *exec3* is more complicated than it first appears. Its call to an auxiliary function, *recurse* hides the detail that each SOS rule enabled in the configuration $(RTN, \sigma_1, \pi_1)$ is executed in a fashion that the state is ramified on conclusion. The function, *recurse* relies on the atomicity of the SOS rules and being non-interfering.

```
91.0  recurse : func-set × RTN-Types × × → ×
  .1  recurse (e, RTN, , ) △
  .2    if e ≠ {}
  .3    then let f ∈ e in
  .4         let mk- ( , ) = cases f :
  .5                          F1 → f1 (RTN, , ),
  .6                          F2 → f2 (RTN, , )
  .7                        end in
  .8         recurse (e \ {f}, RTN, , )
  .9    else mk- ( , )
```

The call to *recurse* in *exec3* is:

$recurse(enabled, RTN, \sigma_1, \pi_1)$

from which *recurse* recursively calls itself until all the *enabled* rules originally specified to it have been executed - this is therefore a greedy semantic. The recursively call with *recurse* is:

$recurse(enabled$
$setf, RTN, \sigma', \pi')$

where $f$ is the function last to be executed which produced the new state $(\sigma')$ and trace $(\pi')$. The choice of which function to execute is made non-deterministically[4] by selecting a function identifier from *enabled*, it is then executed to produce $(\sigma'\pi')$.

[4] Though the VDMToolswill always choose the same element from a set on each execution