Universiteit
Antwerpen

# Translating GPSS to DEVS

## Based on a DEVS Building Block Library

**Randy Paredis**

Promotor: Prof. Hans Vangheluwe
Co-Promotor: Dr. Simon Van Mierlo

Dissertation Submitted in June 2020 to the
Department of Mathematics and Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Ansymo
Antwerp Systems and Software Modelling

# Contents

**List of Figures** v

**List of Tables** vii

**List of Blocks** ix

**Abstract** xi

**Acknowledgements** xii

**Nederlandstalige Samenvatting** xiii

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 3
- 1.3 Structure . . . 3

**2 Background** 4
- 2.1 Modeling Language Engineering . . . 4
  - 2.1.1 Syntax and Semantics . . . 4
  - 2.1.2 World Views . . . 6
- 2.2 DEVS Formalism . . . 7
  - 2.2.1 Classic DEVS . . . 7
  - 2.2.2 Parallel DEVS . . . 10
  - 2.2.3 Simulator . . . 10
  - 2.2.4 Barriers for Non-Programmers . . . 11
  - 2.2.5 Representation of DEVS Building Blocks . . . 11
  - 2.2.6 Push and Pull Systems . . . 13

2.2.7 Time Evaluation in DEVS . . . 13
2.3 GPSS . . . 14
2.3.1 GPSS Syntax . . . 14
2.3.2 Entities . . . 15
2.3.3 Chains . . . 16
2.3.4 Scanning Algorithm . . . 17
2.3.5 Flow . . . 17
2.3.6 Time . . . 18
2.3.7 Resources . . . 19
2.3.8 User Chains . . . 23
2.3.9 Gathering Statistics . . . 24

**3 Tools, Frameworks and Libraries 29**
3.1 Tools . . . 29
3.1.1 ExtendSim . . . 30
3.1.2 SIMUL8 . . . 31
3.1.3 FlexSim . . . 31
3.1.4 Enterprise Dynamics . . . 33
3.1.5 LEGO Mindstorms . . . 33
3.2 DEVS Frameworks and Libraries . . . 34
3.2.1 Python(P)DEVS . . . 34
3.2.2 DEVSimPy . . . 35
3.2.3 JDEVS . . . 35
3.2.4 adevs . . . 36
3.2.5 DEVS++ and DEVS# . . . 36
3.2.6 DEVS-Suite . . . 36
3.2.7 DesignDEVS . . . 36
3.2.8 DEVS-Ruby . . . 37
3.3 GPSS Tools . . . 37
3.3.1 HGPSS . . . 38
3.3.2 GPSS World . . . 38
3.3.3 GPSS/H . . . 38
3.3.4 JGPSS . . . 38
3.3.5 aGPSS . . . 38
3.3.6 AToM$^3$ . . . 39

**4 PythonDEVS-BBL 40**
4.1 Generators . . . 41
4.1.1 Standard Generators . . . 41
4.1.2 Random Number Generators . . . 43
4.1.3 Using Stock . . . 52
4.1.4 Firing Single Events . . . 53

4.1.5 Generating in Bulk . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.2 Queueing Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.2.1 Queueing Theory . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
4.2.2 Building Blocks for Queues . . . . . . . . . . . . . . . . . . . . 58
4.3 Gathering Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.1 Tableization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.3.2 Memory-Efficient Collection . . . . . . . . . . . . . . . . . . . . 65
4.3.3 $P^2$ Algorithm without Scoring Observations . . . . . . . . . 66
4.3.4 Obtaining Numeric Data from other Items . . . . . . . . . . 67
4.3.5 Counting Items . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
4.4 Mathematics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.4.1 General Functionality . . . . . . . . . . . . . . . . . . . . . . . 69
4.4.2 Basic Mathematics . . . . . . . . . . . . . . . . . . . . . . . . . 70
4.4.3 Equations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
4.4.4 Complex Mathematics . . . . . . . . . . . . . . . . . . . . . . . 71
4.5 Input/Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.5.1 Output . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.5.2 Input . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.5.3 Playing Sounds . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.5.4 Listening to External Events . . . . . . . . . . . . . . . . . . . 79
4.6 Transforming Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.6.1 Transforming via Functions . . . . . . . . . . . . . . . . . . . . 80
4.6.2 Lookup Table . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.6.3 Packing and Unpacking . . . . . . . . . . . . . . . . . . . . . . 81
4.7 Routing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.7.1 Exiting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.7.2 Terminating a Simulation . . . . . . . . . . . . . . . . . . . . . 83
4.7.3 Exceptions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
4.7.4 Splitting and Joining . . . . . . . . . . . . . . . . . . . . . . . . 84
4.7.5 Conditionals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
4.7.6 Guards . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.7.7 Gates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.7.8 Time Manipulation . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.7.9 Syncing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.8 Simulation Tracers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
4.8.1 Plot Tracer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
4.8.2 Statistics Tracer . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.8.3 Footprint Tracer . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.8.4 Profile Tracer . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.9 Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
4.10 Implementation Notes . . . . . . . . . . . . . . . . . . . . . . . . . . 99
4.10.1 Library Specifics . . . . . . . . . . . . . . . . . . . . . . . . . . 99

4.10.2 Documentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
4.10.3 Tests . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101

**5 GPSS2DEVS** **102**
5.1 General Principles . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.1.1 Three Principles of GPSS2DEVS . . . . . . . . . . . . . . . 103
5.1.2 Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.2 Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
5.2.1 Entities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
5.2.2 Scanning Algorithm . . . . . . . . . . . . . . . . . . . . . . . 104
5.2.3 Time and Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.2.4 Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
5.2.5 The `TRANSFER` Block . . . . . . . . . . . . . . . . . . . . . 115
5.2.6 User Chains . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.2.7 Gathering Statistics . . . . . . . . . . . . . . . . . . . . . . . 116
5.3 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.3.1 Name Mangling . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.3.2 Python(P)DEVS Formats . . . . . . . . . . . . . . . . . . . . 120
5.4 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121
5.4.1 Manufacturing Shop . . . . . . . . . . . . . . . . . . . . . . . 121
5.4.2 Telephone Exchange . . . . . . . . . . . . . . . . . . . . . . . 121

**6 Wrapping Up** **129**
6.1 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
6.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.3 Further Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
6.3.1 PythonDEVS-BBL . . . . . . . . . . . . . . . . . . . . . . . . 131
6.3.2 GPSS2DEVS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133

**Appendices** **134**

**Appendix A Distributions** **135**
A.1 Distribution Overview . . . . . . . . . . . . . . . . . . . . . . . . . . 135
A.2 Distributions in PythonDEVS-BBL . . . . . . . . . . . . . . . . . . 138

**Appendix B Building Block Port Information** **144**

**Appendix C GPSS Blocks** **159**

**Bibliography** **162**

**Index** **172**

# List of Figures

2.1 Terminology Language Engineering [VVD19; Van+17] . . . . . . . . 5
2.2 Visualization of an Atomic DEVS example [Mal+15] . . . . . . . . . 12

4.1 Plots for the Standard Normal Distribution. . . . . . . . . . . . . . . 45
4.2 Inverse-Transformation with interpolated values. . . . . . . . . . . 48
4.3 Example on how to handle stock with a combination of building blocks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.4 Simple Statechart for a QUEUE. . . . . . . . . . . . . . . . . . . . 59
4.5 An overview of a QUEUE building block with automatic dequeues. . 60
4.6 An example on balking. . . . . . . . . . . . . . . . . . . . . . . . . 62
4.7 Creating a randomized delay from the RANDOM and the TIMER blocks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.8 Example of how to set the *select* input of the CHOOSE OUTPUT block by default. . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.9 Using the GUARD, QUEUE and SINGLE FIRE blocks to create a queue before a critical section. . . . . . . . . . . . . . . . . . . . . 88
4.10 Assigning shifts to sections of the model. . . . . . . . . . . . . . . 89
4.11 Creating the DELAYER building block from the TRANSFORMER and the TIMER. . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.12 Issue with messages arriving at a different time. . . . . . . . . . . 92
4.13 Application to Queueing Systems example model concept [Van14; VV17b]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
4.14 Implementation of *example 4.9.1* in PythonDEVS-BBL, for three processors. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

5.1 The translation of the `GENERATE`, `ADVANCE` and `TERMINATE` blocks in GPSS2DEVS. . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.2 The GPSS2DEVS ADVANCE block as a Coupled DEVS. . . . . . . 109
5.3 *Example 5.2.1*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.4 The translation of the TEST and ASSIGN blocks in GPSS2DEVS. . 111
5.5 The translation of the SEIZE and RELEASE blocks in GPSS2DEVS. 112
5.6 The translation of the ENTER and LEAVE blocks in GPSS2DEVS. . 113
5.7 The translation of the LOGIC and GATE blocks in GPSS2DEVS. . . 114
5.8 The GPSS2DEVS TRANSFER block in *conditional* mode as a Coupled DEVS. . . . . . . . . . . . . . . . . . . . . . . . . . . 115
5.9 The translation of the TRANSFER block in *conditional* mode, combined with the SEIZE and RELEASE blocks in GPSS2DEVS. . . . . 116
5.10 The translation of the LINK and UNLINK blocks in GPSS2DEVS. . 117
5.11 The translation of the MARK and TABULATE blocks in GPSS2DEVS. 118
5.12 The translation of the QUEUE and DEPART blocks in GPSS2DEVS. . 119
5.13 Visual Notation in AToM$^3$ for *Example 5.4.1* (the manufacturing shop). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
5.14 Textual Notation for *Example 5.4.1* (the manufacturing shop). . . . 123
5.15 Python(P)DEVS code for *Example 5.4.1* (the manufacturing shop). 124
5.16 Visual Notation in AToM$^3$ for *Example 5.4.2* (the telephone exchange). . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
5.17 Textual Notation for *Example 5.4.2* (the telephone exchange). . . . 127
5.18 Python(P)DEVS code for *Example 5.4.2* (the telephone exchange). 128

# List of Tables

2.1 GPSS "punchcard" fields. . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Overview of the gathered statistics in GPSS by static entities. . . . 26
2.3 Input for *example 2.3.1*. . . . . . . . . . . . . . . . . . . . . . . . . 26
2.4 Gathered Statistics for *example 2.3.1*. . . . . . . . . . . . . . . . . 26

3.1 Default building blocks in DEVSimPy. . . . . . . . . . . . . . . . 35
3.2 Default building blocks in DEVS-Ruby. . . . . . . . . . . . . . . . 37

4.1 Closed-formula form of CDFs (and their inverse). . . . . . . . . . . 47
4.2 The facility code table for syslog messages. . . . . . . . . . . . . . 75
4.3 The severity code table for syslog messages. . . . . . . . . . . . . 76

5.1 Obtained metrics for *example 5.4.1*. . . . . . . . . . . . . . . . . . . 122
5.2 Obtained metrics for *example 5.4.2*. . . . . . . . . . . . . . . . . . . 125

B.1 Ports for the Constant Generator building block. . . . . . . 144
B.2 Ports for the Function Generator building block. . . . . . . . 145
B.3 Ports for the Table Generator building block. . . . . . . . . . 145
B.4 Ports for the Single Fire building block. . . . . . . . . . . . . . 145
B.5 Ports for the Bulk Generator building block. . . . . . . . . . 145
B.6 Ports for the Random Number Generator building block. . . . 146
B.7 Ports for the Random Delay Generator building block. . . . 146
B.8 Ports for the Simple Queue building block. . . . . . . . . . . . . 147
B.9 Ports for the Queue Tracker building block. . . . . . . . . . . 147
B.10 Ports for the Queue building block. . . . . . . . . . . . . . . . . . 148
B.11 Ports for the Retain building block. . . . . . . . . . . . . . . . . . 149
B.12 Ports for the Advance building block. . . . . . . . . . . . . . . . . 149
B.13 Ports for the Table Collector building block. . . . . . . . . . 149

B.14 Ports for the Collector and Estimate Collector building blocks. . . . 149
B.15 Ports for the Counter building block. . . . 150
B.16 Ports for the Adder building block. . . . 150
B.17 Ports for the Multiplier building block. . . . 150
B.18 Ports for the Equation building block. . . . 151
B.19 Ports for the Differentiator building block. . . . 151
B.20 Ports for the Integrator building block. . . . 151
B.21 Ports for the Random building block. . . . 151
B.22 Ports for the Logger building block and all its derivatives. . . . 152
B.23 Ports for the File Writer building block. . . . 152
B.24 Ports for the File Reader building block. . . . 152
B.25 Ports for the Listener building block. . . . 152
B.26 Ports for the Sound building block. . . . 152
B.27 Ports for the Transformer and Lookup Table building blocks. 153
B.28 Ports for the Pack building block. . . . 153
B.29 Ports for the Unpack building block. . . . 153
B.30 Ports for the Finish and Halt building blocks. . . . 153
B.31 Ports for the Choose Input building block. . . . 154
B.32 Ports for the Choose Output building block. . . . 154
B.33 Ports for the Pick building block. . . . 154
B.34 Ports for the Guard building block. . . . 155
B.35 Ports for the Gate building block. . . . 155
B.36 Ports for the Timer building block. . . . 155
B.37 Ports for the Delayer building block. . . . 156
B.38 Ports for the Sync building block. . . . 156
B.39 Ports for the Controller building block. . . . 157
B.40 Ports for the Hold building block. . . . 158

# List of Blocks

2.1 General depiction of a DEVS block within this paper. . . . . . . . . . 13
2.2 GPSS GENERATE block. . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.3 GPSS TERMINATE block. . . . . . . . . . . . . . . . . . . . . . . . . 18
2.4 GPSS TRANSFER block. . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.5 GPSS ASSIGN block. . . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.6 GPSS TEST block. . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.7 GPSS ADVANCE block. . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.8 GPSS SEIZE and RELEASE blocks. . . . . . . . . . . . . . . . . . . . 21
2.9 GPSS PREEMPT and RETURN blocks. . . . . . . . . . . . . . . . . . . 22
2.10 GPSS ENTER and LEAVE blocks. . . . . . . . . . . . . . . . . . . . . 23
2.11 GPSS LOGIC and GATE blocks. . . . . . . . . . . . . . . . . . . . . . 24
2.12 GPSS LINK and UNLINK blocks. . . . . . . . . . . . . . . . . . . . . 25
2.13 GPSS TABULATE block. . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.14 GPSS MARK block. . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.15 GPSS QUEUE and DEPART blocks. . . . . . . . . . . . . . . . . . . . 28

4.1 Constant Generator building block. . . . . . . . . . . . . . . . 42
4.2 Function Generator building block. . . . . . . . . . . . . . . . 42
4.3 Table Generator building block. . . . . . . . . . . . . . . . . . 43
4.4 Random Number Generator building block. . . . . . . . . . . 52
4.5 Random Delay Generator building block. . . . . . . . . . . . 52
4.6 Single Fire building block. . . . . . . . . . . . . . . . . . . . . 54
4.7 Bulk Generator building block. . . . . . . . . . . . . . . . . . . 54
4.8 Simple Queue and Queue building blocks. . . . . . . . . . . . 60
4.9 Queue Tracker specialized building block. . . . . . . . . . . . 61
4.10 Retain building block. . . . . . . . . . . . . . . . . . . . . . . . 62
4.11 Advance building block. . . . . . . . . . . . . . . . . . . . . . . . 64

4.12 Table Collector building block. . . . . . . . . . . . . . . . . . 65
4.13 Collector building block. . . . . . . . . . . . . . . . . . . . . . . . 66
4.14 Estimate Collector building block. . . . . . . . . . . . . . . . 67
4.15 Counter building block. . . . . . . . . . . . . . . . . . . . . . . . 69
4.16 Adder building block. . . . . . . . . . . . . . . . . . . . . . . . . 70
4.17 Multiplier building block. . . . . . . . . . . . . . . . . . . . . . 70
4.18 Equation building block. . . . . . . . . . . . . . . . . . . . . . . 71
4.19 Differentiator building block. . . . . . . . . . . . . . . . . . . 72
4.20 Integrator building block. . . . . . . . . . . . . . . . . . . . . . 72
4.21 Random building block. . . . . . . . . . . . . . . . . . . . . . . . 73
4.22 Representation of the Logger building block and its derivatives. . 77
4.23 File Writer building block. . . . . . . . . . . . . . . . . . . . . 77
4.24 File Reader building block. . . . . . . . . . . . . . . . . . . . . 78
4.25 Sound building block. . . . . . . . . . . . . . . . . . . . . . . . . 79
4.26 Listener building block. . . . . . . . . . . . . . . . . . . . . . . . 79
4.27 Transformer building block. . . . . . . . . . . . . . . . . . . . . 80
4.28 Lookup Table building block. . . . . . . . . . . . . . . . . . . . 81
4.29 Pack building block. . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.30 Unpack building block. . . . . . . . . . . . . . . . . . . . . . . . 82
4.31 Finish building block. . . . . . . . . . . . . . . . . . . . . . . . . 83
4.32 Halt building block. . . . . . . . . . . . . . . . . . . . . . . . . . 83
4.33 Choose Input and Choose Output building blocks. . . . . . 85
4.34 Pick building block. . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.35 Guard building block. . . . . . . . . . . . . . . . . . . . . . . . . 87
4.36 Gate building block. . . . . . . . . . . . . . . . . . . . . . . . . . 89
4.37 Timer building block. . . . . . . . . . . . . . . . . . . . . . . . . 90
4.38 Delayer building block. . . . . . . . . . . . . . . . . . . . . . . . 91
4.39 Sync building block. . . . . . . . . . . . . . . . . . . . . . . . . . 92

5.1 GPSS2DEVS Controller and Hold blocks. . . . . . . . . . . 105

# Abstract

Models are everywhere. From simple mathematical equations to a schematic representation of a water filtering plant to a waiting line at the cash registry in a super market. They're often used to get a solid understanding of business flows, system flows, proof-of-concepts, factories...

Different modeling languages within the domain of Modeling and Simulation (M&S) can be used for various purposes. A subset of these languages describes discrete-event systems and can be further separated based on their world view: event scheduling, activity scheduling or process interaction.

This thesis will use the modeling formalism of Discrete EVent Systems (DEVS), a general-purpose event-scheduling language, to create a generic building block library (BBL). Multiple tools, that are currently used for big systems by major companies, will be explored as a foundation for this library.

Additionally, the General Purpose Simulation System (GPSS), an example language in the process interaction world view, will be described and a subset thereof will be translated onto the DEVS formalism. This way, we benefit from the advantages that DEVS offers, while still enabling the strengths of GPSS.

Accompanied with this paper, there is a Python-implementation (based on the Python(P)DEVS kernel) of the building blocks and the translation that will be discussed in these pages.

# Acknowledgements

Firstly, I would like to thank my promotor, Hans Vangheluwe, for mentoring me and guiding me throughout my internship and master thesis. He helped me find a passion and interest in simulation and modeling by giving me the opportunity to work on fascinating research projects.

I also want to thank my co-promotor, Simon Van Mierlo, for guiding me in the right direction by providing enough resources, ideas and much appreciated feedback that helped the research.

My final thanks go out to my friends and family that have supported me and kept me sane throughout my entire study career.

# Nederlandstalige Samenvatting

Modellen zijn overal. Van eenvoudige wiskundige vergelijkingen tot schematische voorstellingen van waterfilterinstallaties tot een wachtrij aan de kassa in de supermarkt. Ze worden vaak gebruikt om een diepgaand overzicht te krijgen van bedrijfsprocessen, systeemstromen, bewijzen van concepten, fabrieken...

Verschillende modelleertalen binnen het domein van Modellering en Simulatie (M&S) kunnen voor verscheidene doeleinden worden gebruikt. Een subset van deze talen beschrijft discrete evenementsystemen en kan verder worden opgesplitst op basis van hun world view: ofwel evenementplanning, activiteitenplanning of procesinteractie.

Deze thesis zal het modeleerformalisme van Discrete EVent Systems (DEVS), een algemene taal voor evenementplanning, gebruiken om een generische bouwblokkenbibliotheek te maken. Verscheidene programma's, die momenteel gebruikt worden door grote bedrijven voor enorme systemen, zullen worden onderzocht als een basis voor deze bibliotheek.

Daarbovenop zal het General Purpose Simulation System (GPSS), een voorbeeld van een taal in de procesinteractie world view, worden beschreven en een subset ervan zal worden vertaald naar het DEVS formalisme. Hiermee genieten we van de voordelen die DEVS aanbiedt en behouden we tezelfdertijd de sterktes van GPSS.

Bijgevoegd aan deze paper is er een Python-implementatie (gebaseerd op de Python(P)DEVS kernel) van de bouwblokken die in deze pagina's zullen worden besproken, net als de benodigde code voor de vertaling van GPSS naar DEVS.

# CHAPTER 1

# Introduction

Be it a waiting line at your local supermarket, or a conveyor belt in a factory. Be it a telephone switchboard, or a traffic light at the crossroads of train tracks. Complex systems are everywhere. In an ideal scenario, the analysis of such systems should happen in an efficient way.

*Models* can be used to provide an understandable and complete overview of the problem domain. Ideally, these models can be interpreted at numerous levels of domain-expertise. Next, *simulations* are executed on these models to gather information about the system within a certain context.

Besides a motivation for this research, this chapter will also state the contributions that this thesis will make. Furthermore, the general structure of this document will be defined.

## 1.1 Motivation

Massive software-intensive and cyber-physical systems often become complex quite fast, either when designing them, or when they are deployed. Even trying to study such systems requires a lot of surrounding knowledge that might not be widely available. For instance, in designing a system that needs to interact with real-world scenarios (e.g. a car, a plane...), some of the required physics and knowledge might only be available as a consequence of numerous experiments, from which certain conclusions can be used in the system.

The field of Modeling and Simulation (M&S) tries to overcome the complexities that such systems cause, for the benefit of the engineers. M&S makes

use of so-called *modeling languages* (or *formalisms*) that describe these systems, both in *syntax* (what is and isn't allowed in the language and how it can be represented) and the *semantics* (the meaning of the models in the language) [VVD19; Van+17]. Multiple formalisms can be used to identify different characteristics of a system.

Most of these languages are *domain-specific* [Ris10], meaning that they can be used to solve problems within a particular domain. Yet, the creation of such a domain-specific language can be cumbersome. A better idea is to use a *general-purpose language* (that is not linked to a particular domain) to create *libraries* of reusable components that can be used within (but are not limited to) the domain in question.

Multi-Paradigm Modeling (MPM) [MV04; MV14; Syr11; GLV07] encourages the use of the most appropriate language(s) for the most appropriate level(s) of abstraction. The area of *multi-formalism modeling* allows the combination of multiple languages for this very purpose [Van08]. Furthermore, domain experts can make use of (visual) modeling notations they are familiar with, avoiding so-called *accidental complexity* when there is too much of a gap between the domain expert's knowledge and the modeling notation.

Different modeling languages can be separated into multiple categories, which help us analyze and study them. This paper will mainly focus on the *discrete-event* modeling languages, which have a sub-categorization of three distinct world views: *event scheduling*, *activity scanning* and *process interaction* [ON04]. A translation between these world views is non-trivial because implicit assumptions of a world view need to be made explicit in other world views.

[Van00a; Van00b] introduce the notion that the Discrete EVent Systems (DEVS) formalism [ZPK00], an example of the event scheduling world view, can be used as a *common denominator* for discrete-event simulation formalisms. This means that it can be used to implement the semantics of many modeling languages and, as such, their simulators can be coupled.

Hence, it is an interesting and relevant research trying to translate different formalisms onto DEVS. While the loss of some information in the translation may be inevitable (due to major differences between formalisms), we can at least try to identify a *model transformation* [SK03] that causes the least amount of loss.

The syntax of discrete-event languages might state the usage of *building blocks*. These can be seen as some sort of LEGO bricks that fit closely together in the creation of a model. Each type of brick has its own functionality and may react on input, or send out custom outputs, to be interpreted by the user or other building blocks.

Making use of a Python implementation for DEVS, Python(P)DEVS [Van14; VV16], we will construct a library of reusable building blocks, focusing on

*extensiveness* (each block can be used in many models of many domains) and *deficiency* (the omission of a block is an indication of the rarity of its use).

Next, small parts of the building block library will be used to translate GPSS [Gor78b; Gor78a; Sch74], a representative example of the process interaction world view, onto DEVS.

## 1.2 Contributions

In this thesis, ExtendSim [Ima87], SIMUL8 [SIM94], FlexSim [Fle93], Enterprise Dynamics [INC97] and LEGO Mindstorms [LEG13] will be studied to get an overall understanding of existing building block libraries within major tools that are used in industrial (and less industrial) contexts. We will focus on the functionalities they provide, how they are structured and to what extend they correspond to the DEVS formalism.

Additionally, existing DEVS modeling tools (e.g. JDEVS [FB04], DEVSimPy [Cap19; Cap+11], DEVS-Suite [KSE09], DesignDEVS [GBK16]...) will be placed under the microscope w.r.t. their building block libraries, from which we can deduce what's already available.

Given this knowledge, we'll start creating our own building block library, based on Python(P)DEVS, including all that's already provided (by other tools) and all that is common in professional modeling tools for discrete-event simulation. Finally, we'll take a look at a subset of GPSS and provide a mapping onto DEVS. This will be given such that every GPSS block has at least one corresponding block in the translated DEVS model.

## 1.3 Structure

*Chapter 2* will introduce a background for the remainder of this thesis, including a description of the DEVS and the GPSS formalisms. In *chapter 3*, multiple discrete-event tools that are used in industry are studied. We will also discuss tools to model in DEVS and GPSS. With this information, a general idea can be constructed for what we require in the creation of a new building block library for Python(P)DEVS, which will be built in *chapter 4*. Next, *chapter 5* will go into detail on a translation from DEVS to GPSS in such a way that the original models can be extracted again from their translated version. Finally, *chapter 6* will list some conclusions of this thesis, as well as some further work to be done on this topic.

# CHAPTER 2

# Background

It is important to be aware of the basis of research. Hence, this chapter will provide some information on *modeling language engineering* in *section 2.1*. *Sections 2.2* and *2.3* will introduce the two main formalisms that are used in this thesis, namely DEVS and GPSS.

## 2.1 Modeling Language Engineering

An overplus of modeling languages are used by numerous simulationists in many different problem domains. Multi-Paradigm Modeling (MPM) [MV04; MV14; Syr11; GLV07] advocates modeling every relevant aspect of the system explicitly at the most appropriate level(s) using the most appropriate formalism(s). *Multi-formalism modeling* reinforces this concept by allowing the combination of multiple modeling languages within a single model [Van08].

### 2.1.1 Syntax and Semantics

Obviously, we cannot start creating modeling languages "at random". Each formalism needs to be precisely defined, with a *syntax* and a *semantics*. [Kle07] states that a modeling language needs to be fully defined by:

- Its *abstract syntax*, which defines the constructs and combinations thereof that may occur. It describes what is (or isn't) allowed by a modeler using this language. Often a *meta-model* [Küh06] is used to capture these constraints.

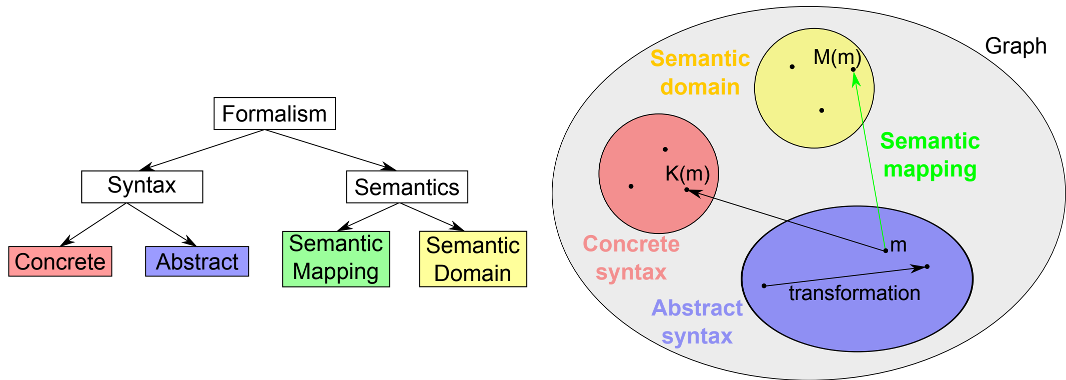



Figure 2.1: Terminology Language Engineering [VVD19; Van+17]

- Its *concrete syntax*, which specifies the representation of the defined constructs. This can be either textual or graphical (using icons).
- The *semantics*, which defines the meaning of the models that are created [HR04]. This is comprised of a *semantic domain* (i.e. *what* the models mean) and the *semantic mapping* (i.e. *how* the models in the language can be interpreted). The semantic domain can be seen as the target domain of the semantic mapping.

For instance, $(3+4)$, $(+\ 3\ 4)$ and $(3\ 4\ +)$ are all a textual concrete syntax for the concept (i.e. abstract syntax) of "the sum of 3 and 4". The semantic domain for this example is the set of natural numbers, whereas the semantic mapping (i.e. the meaning) is 7 $(= 3 + 4)$.

*Figure 2.1* shows a definition of the above terminology, as per [VVD19; Van+17]. Each aspect of a formalism is modeled explicitly, as well as relationships between different formalisms.

Furthermore, there are two types of semantics we can distinguish between:

- The *translational semantics*, i.e. the semantics for translating a model in one formalism into a model that follows another in such a way that the resulting model is equivalent to the original model w.r.t. the properties under study.
- The *operational semantics*, i.e. the semantics that define the functionalities of the model. Often this can be seen as an *execution* or a *simulation* of the model.

A language that coheres to these rules and is viable within a specific modeling domain is said to be a *domain-specific language* (DSL) [Ris10]. Yet, the creation of new DSLs has downsides.

- For one, it is a major investment to create new languages that are useful within a specific domain, while at the same time being usable by domain experts.
- It is important that the creation of a new DSL does not supersede other, more prominent languages [Bar+11].
- Certain constructs in the newly created DSL may be useful within other domains. And, while the other domains could make use of the new DSL, there might also be constructs in the DSL that cannot be applied within the context of these other domains.

We can solve this latter issue with sets of reusable artifacts (i.e. *libraries*) for a *general-purpose language* (that is not tied to a specific domain). In this case, no new language needs to be created and the library can be specialized for the problem domain.

With the existence of countless different modeling languages, most of whom DSLs, we require a way to classify them into categories that we can reason about.

### 2.1.2 World Views

A time base is a way of ordering the observations in a system [ZPK00]. This can be *logical time*, i.e. the abstract order of events that simulate the actual time; or the *physical time*, i.e. the real-time (or wall-clock time) in which the system functions [Ley20]. Depending on the context of your modeling language, one is preferred above the other. And it is within this context that a first distinction can be made for classifying the modeling languages [ZPK00].

On the first axis, there are *discrete-time* modeling languages. They assume that the state of the system changes at discrete (often equidistant) points in time. In between these intervals, the system is not defined. Perpendicular to discrete-time languages, there are the *discrete-event* modeling languages. Here, models have a specific, discrete state that can change in every time step, but remains constant in between these steps. Think, for instance, about a light bulb that can be on or off. Finally, *continuous-time* systems have a state which changes throughout time. In this case, the light bulb does not have an on/off switch, but rather a dimmer that gradually increases the light's intensity.

Discrete-event systems (what we'll be focusing on in this thesis) can be separated even further by using multiple *world views* [ZPK00]. [ON04] identifies three such world views, all of whom attempt to capture different notions of locality:

*Event Scheduling* provides the *locality of time*: each event routine in a model specification describes related actions that should always occur in one instant.

*Activity Scanning* provides the *locality of state*: each activity routine in a model specification describes all actions that should occur due to the model assuming a particular state (that is, due to a particular condition becoming true).

*Process Interaction* provides the *locality of object*: each process routine in a model specification describes the action sequence of a particular model object.

[ON04] also discusses relationships between these world views and examines if they can be transformed into one another. Unfortunately, such a transformation is complicated because implicit assumptions of a world view need to be made explicit in other world views.

## 2.2 DEVS Formalism

Discrete-EVent Systems (DEVS) [ZPK00] is actually a pretty elegant and clean formalism that can be used as a *common denominator* [Van00a] for numerous other formalisms like Causal Block Diagrams (CBD) [GDV16], Statecharts (SC) [Har87; BV03; SV11] and Petri Nets (PN) [Pet77; Mur89], as stated in [VVV18; Van00a; Van00b]. Nevertheless, there are dozens of variants for this formalism (like PDEVS[CZ94], Dynamic Structure DEVS [Bar95; Bar97; Bar98; Uhr01], Fuzzy-DEVS [Kwo+96]...), all of whom with interesting aspects and peculiarities, but seeing as most of these are mere extensions of the basic DEVS formalism, we will assume these formalisms out-of-scope for the purposes of this paper. Instead, we will be focusing Classic DEVS (the most basic form), as described in [ZPK00]. The interested reader can extend the information in these pages towards the DEVS formalism of their liking.

### 2.2.1 Classic DEVS

Classic DEVS (CDEVS) was first described in [Zei84]. Afterwards, many have summarized and referenced his work, allowing them to make extensions and adaptations on it. But in order for this formalism to align itself to the intends and purposes of this paper and, furthermore, to allow the existing extensions to be applicable on the descriptions that will be given further on, it's ought to be important to return to the basics. In its most simple form, CDEVS consists of Atomic DEVS that can be coupled together in Coupled DEVS for more complex structures.

### Atomic DEVS

The smallest self-supporting element in a model can be called an *atom*. Therefore, the DEVS representation of such an *atom* will go by the name "Atomic DEVS". These *atoms* cannot be further separated into smaller parts, because its functionality would disintegrate. Similarly to the atoms in chemistry, that, until the discovery of subatomic particles in 1897, were believed to be the smallest possible particles in the universe. Hence, the name "atom", which is derived from the Greek $\alpha\tau o\mu o\varsigma$[1], meaning "indivisible".

An Atomic DEVS model itself is defined as the 7-tuple

$$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where $X$ denotes the *input set* and $Y$ the *output set*. Both of them can contain multiple ports, making both $X$ and $Y$ cartesian products of their respective ports, i.e.

$$X = \times_{i=1}^{m} X_i \qquad\qquad Y = \times_{j=1}^{l} Y_j$$

with each $X_i$ the admissible inputs on port $i$ and each $Y_j$ the admissible outputs on port $j$. Hence, $X$ is the set of all possible inputs and $Y$ the set of all possible outputs of the model.

$S$ is the *state set*, meaning $S$ is the structured set that contains all the possible states $M$ can be in, with $s_0 \in S$ the *initial state*. Do note that we will slightly adapt the original formalism and make this state explicit [VV18].

$$S = \times_{i=1}^{n} S_i$$

The *internal transition function*, $\delta_{int}$, is a function that defines the next state, based on the current state. This function will be called when the DEVS block decides it's time to act. This is decided via $ta$, better known as the *time advance function*. Based on its current state, $ta$ yields a positive, real number (or $+\infty$), identifying the amount of time to wait until the next internal event[2].

$$\delta_{int} : S \to S \qquad\qquad ta : S \to \mathbb{R}^{+}_{+\infty}$$

Notice that, when $ta$ equals $+\infty$, we are in a so-called *passive state*, meaning that an internal transition will never happen.

$\delta_{ext}$, the *external transition function*, is called every time we obtain an input. Based on this input ($\in X$), the current state $s$ and the elapsed time $e$, $\delta_{ext}$ determines a new state for $M$. After $\delta_{ext}$ is called, $ta$ is called once more to reschedule the previously scheduled internal event.

$$\delta_{ext} : Q \times X \to S \qquad\qquad Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$$

[1] The $\alpha$ usually has a double accent, which was omitted for readability.

[2] $+\infty$ identifies that no new event is scheduled.

Finally, we have $\lambda$, the *output function*. Based on the current state of $M$, $\lambda$ returns an output ($\in Y$) or the so-called *null event* $\phi$.

$$\lambda : S \rightarrow Y \cup \{\phi\}$$

### Coupled DEVS

When we start combining Atomic DEVS together in a network $\Delta$, we obtain a Coupled DEVS. It stands to reason that a Coupled DEVS not only allows for hierarchy and embedding, but also enables the modeler to divide their problem into smaller sub-problems, yielding quite a powerful formalism.

Similar to Atomic DEVS, a Coupled DEVS model is a 7-tuple, but it has a slightly different definition to allow for the coupling.

$$M = \langle X_\Delta, Y_\Delta, D, M_i, I_i, Z_{i,j}, select \rangle$$

Here, $X_\Delta$ and $Y_\Delta$ respectively represent the *input* and *output set* of network $\Delta$, while $D$ are the *component references* that make up the Coupled DEVS. Next, $M_i$ is the *set of all sub-components*, i.e. $M_i$ is the Atomic DEVS model for component $i$, for all $i \in D$.

The *set of influences*, $I_i$ is the set of all blocks whose inputs are linked to $i$'s output port(s). $Z_{i,j}$ represents the *transfer functions*. They are applied to all messages being passed and allow for reuse, due to their compatibility features. For all $i \in D \cup \{\Delta\}$ and $j \in I_i$:

$$Z_{\Delta,j} : X_\Delta \rightarrow X_j$$
$$Z_{i,\Delta} : Y_i \rightarrow Y_\Delta$$
$$Z_{i,j} : Y_i \rightarrow X_j$$

At last, *select* is the *tie-breaking function*, which is used to resolve collisions between multiple components of the model. Such a collision occurs if two components schedule their $\delta_{int}$ at the same time.

$$select : 2^D \rightarrow D$$

### Closure under Coupling

As is proven in [ZPK00], DEVS are closed under coupling. This means that, for every Coupled DEVS, an equivalent Atomic DEVS can be constructed. So, instead of only allowing a network of Atomic DEVS to construct a Coupled DEVS, we can generalize this rule by stating that a Coupled DEVS may also consist of other Coupled DEVS.

Turning a Coupled DEVS into its Atomic counterpart is called *flattening*, for which an efficient algorithm is given in [CV10]. Another way to specify

the semantics of Coupled DEVS is by defining pseudocode for the simulation (as was done for Atomic DEVS) [ZPK00]. A performance-oriented view thereof can be found in [Nut10] and, additionally, an example-driven introduction to DEVS and its abstract simulator is presented in [Wai09].

### 2.2.2 Parallel DEVS

The CDEVS formalism was extended to Parallel DEVS (PDEVS) in [CZ94]. Nowadays, most simulators use PDEVS instead of CDEVS.

Instead of using a *select* function to choose which $\delta_{int}$ to execute, PDEVS will allow all imminent components to execute their internal transitions at the same time (hence "parallel"). Obviously, this also means that their $\lambda$ should also be executed, creating multiple outputs. In their turn, these multiple outputs need to be routed to multiple external transition functions. PDEVS makes use of *bags* to allow this routing. A bag can be seen as a collection of messages. All inputs for a specific port are then gathered in such a bag, before being passed onto the corresponding $\delta_{ext}$.

Now, whenever an Atomic DEVS model receives a bag of inputs at the same time as it has scheduled an internal transition function, we don't know what should happen first. The *confluent transition function* $\delta_{con}$ solves this ambiguity by allowing the user to specify what happens in this case. By default, $\delta_{int}$ is called before $\delta_{ext}$.

The formal description for an Atomic DEVS in PDEVS therefore becomes:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$$

For the Coupled DEVS, we can identify the removal of the *select* function:

$$M = \langle X_\Delta, Y_\Delta, D, M_i, I_i, Z_{i,j} \rangle$$

[CZ94] also proves the closure under coupling in this case.

### 2.2.3 Simulator

[MN05] describes an abstract simulator for DEVS. Tools for modeling in the formalism, like Python(P)DEVS [Van14] and adevs [Nut15], implement these semantics.

A simulation step for CDEVS can be summarized as follows:

1. Compute the *imminent components*, i.e. the Atomic DEVS models that have a $\delta_{int}$ that's scheduled to fire.

2. Use the *select* tie-breaking function to choose one of these models.

3. Generate output via executing its $\lambda$ function.

4. Send the outputted events to the correct inputs.

5. For the imminent component, execute its $\delta_{int}$. In parallel, all $\delta_{ext}$ functions will be executed for the Atomic DEVS that receive events.

6. Compute the $ta$ for each Atomic DEVS model.

### 2.2.4 Barriers for Non-Programmers

At Autodesk, they found that the standard DEVS specification was quite difficult to understand for inexperienced users [Mal+15]. More specifically, the separation between the model and the simulation can become quite abstract, especially if a simulator hides this functionality behind a visually appealing user interface.

Autodesk set out to try and solve this issue, making use of ideas from *Human-Computer Interaction*. The result is an Atomic DEVS[3], as is shown in *figure 2.2*.

The Atomic DEVS is initialized in the *Initialization* step by a set of parameters, yielding an initial state ($s_0$). From there, it waits until some time is passed ($ta$), but it can be interrupted by an incoming message in a so-called *Unplanned Event* ($\delta_{ext}$), or by the end of the simulation, in which case we go to a *Finalization* step where statistics can be gathered from the block. The *Unplanned Event* will restart the waiting time after it is done processing. When no interruption occurs, the block may finish waiting and go to a *Planned Event* ($\delta_{int}$). Output may be generated from the current state ($\lambda$)[4] and afterwards, we restart the timer for the next *Planned Event*.

### 2.2.5 Representation of DEVS Building Blocks

In order to create complex DEVS models (mainly in the context of combined building blocks), these pages will denote each block with a unique, graphical representation [Moo09], loosely based on [Tra09]. To explain this notation, an example is given in *block 2.1*.

The blocks are defined by a few properties, as described below. These are not absolute truths, but more of a guideline that will be followed throughout the remainder of this thesis.

*The input and output ports.* These represent the $X$ and $Y$ sets, as described in *section 2.2.1*. For the example given, there are three input ports (*in1*, *in2* and *in3*); and three output ports (*out1*, *out2* and *out3*). An input is differentiated from an output by the directionality of the triangle. Usually, input ports are on the left and output ports are on the right,

[3] Within the context of a banking example.

[4] Note that the *Planned Event* is a representation of both $\delta_{int}$ and $\lambda$.

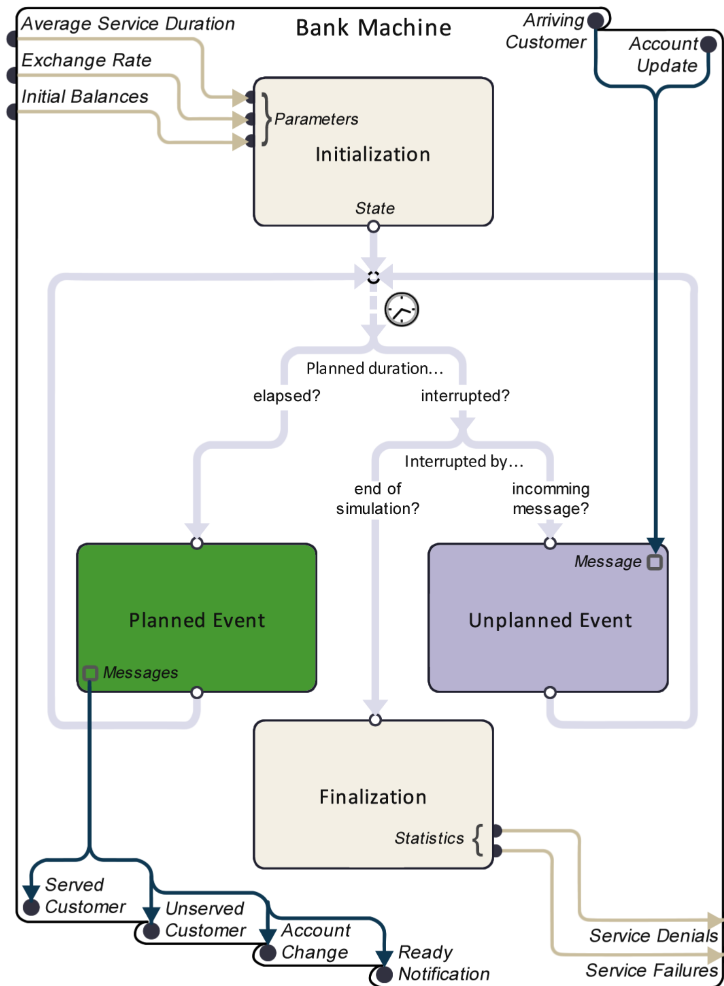



Figure 2.2: Visualization of an Atomic DEVS example [Mal+15]

but depending on the block type and how it's connected in a network, the locations of the ports may change to increase readability. *Appendix B* contains an overview of the descriptions for all building blocks that this thesis will introduce.

*The color.* The background color of the blocks allow us to group them together by topic. Those that have issues with differentiating between different colors can still rely on the other properties defined on these blocks. In *chapter 4*, it will become clear which category has which color. We'll use a light gray to indicate an uncategorized block.

*The type.* Each block has a specific type. This represents the actual functionality of the block in question.

Block 2.1: General depiction of a DEVS block within this paper.

*The name.* The name for the block. This can be used to distinguish between multiple blocks of the same type. Whenever such a distinction is unnecessary, it will be omitted.

Of course, this is just a denotation used within the pages of this paper and does not represent a must-use representation.

### 2.2.6 Push and Pull Systems

When talking about *push* and *pull* systems, we are talking about how the messages are being sent. In a push system, the building blocks output a message as soon as they're done with it. On the other hand, in a pull system, each block "sucks" the message from the previous block.

An advantage of push systems over pull systems is that the message only moves from block to block if it has been processed fully. A disadvantage is that you sometimes want to indicate you're ready for new inputs, which is where pull systems enter the stage.

#### Combined Systems

Sticking to either push or pull can be constrictive, hence a lot of systems allow both to work in harmony. Each block is commonly given a special kind of port that works for pulling and another that handles pushing.

Because DEVS are clearly push systems, it is pertinent we can transform a push system into a pull system. This is a construction that will be used in a lot of building blocks discussed further on and allows us to work with DEVS as if they were a combined system. Instead of having a single in- and output port, we will use some sort of "request" system. Every time we want to pull a message, we request the previous block to push the message through. Hence, we will introduce a *request* output and a corresponding *receive* input, for each pull port.

### 2.2.7 Time Evaluation in DEVS

CDEVS are evaluated at each event, not at each time instance, meaning that there can be multiple events happening at the same time. The *select* function

distinguishes an order in which the CDEVS models may fire their $\delta_{int}$.

Now, when we have a model $A$ that's connected to a model $C$ and a model $B$ that's connected to $C$ and both $A$ and $B$ have an internal transition function scheduled at time $t$, it is dangerous to assume that $C$ will receive $A$'s output and $B$'s output at the same function call for $\delta_{ext}$.

For the purposes of this paper, we will call each firing of a $\delta_{int}$ a *timeframe*. If $A$ and $B$ have a $\delta_{int}$ scheduled at the same time, the timeframe belonging to $A$ will be executed before the timeframe of $B$'s $\delta_{int}$ fires (or vice-versa). Each point in time can therefore consist of multiple timeframes. This notion of time is also known as *superdense time* [SS15].

While this issue does not occur in PDEVS, it is pertinent to keep this in mind when parallelism is not used. See *section 4.7.9* for more information on this issue and on how to prevent/bypass it.

## 2.3 GPSS

The General Purpose Simulation System (GPSS) [Gor78b; Gor78a; Sch74; Gor75], originally named "Gordon's Programmable Simulation System"[5], was created by Gordon at IBM in the 1962. It is a discrete event formalism and while it has made way for more modern simulation and programming approaches, it is still a powerful modeling language. Besides being a programming language, GPSS is also a run-time environment for building and simulating discrete-event models. After conception, multiple versions were introduced. The two most popular ones are GPSS/H and GPSS V.

In order to show, as a proof-of-concept that GPSS can be translated to DEVS, we will only be discussing a subset of GPSS V. The interested reader can extend the provided functionality to cover the full formalism. [Gor75] provides a full description of GPSS V and can be used for this purpose.

### 2.3.1 GPSS Syntax

In GPSS, you create models that you will simulate. These models fully follow a textual concrete syntax. Additionally, users can also make use of a graphical concrete syntax, that fully encompasses the same abstract syntax, but provides a more visual representation of the models. Yet, finding a tool that provides the operational semantics for these graphical models is rare (at the time of writing).

Similarly to DEVS, GPSS makes use of building blocks that are connected in a *block diagram*. For each of these building blocks, we'll provide both

[5] The name changed when the software was released.

| Position | Description |
|---|---|
| 1 | A * in this field indicates that the line is a comment. |
| 2-6 | Depending on the statement, this is either a required or positional label. |
| 8-18 | Operation to be executed. |
| 19-71 | Space-separated parameters, possibly followed by additional comments. |
| 72-80 | Not used and often omitted. |

Table 2.1: GPSS "punchcard" fields.

the textual and the graphical concrete syntax. To identify GPSS constructs (like building blocks, parameters, entities...), we will use a `typewriter font` throughout these pages.

**Statements**

In general, GPSS follows the old *punchcard* construct, where every line consists of 80 characters. *Table 2.1* lists the definitions of the fields as per [Cla92] and [Gor75]. Every statement fits in exactly one line.

When a line starts with an *asterisk* (`*`), it indicates that the line is a comment. If required, a positional label can be added in positions 2-6 and positions 8-18 contain the operation that we want to execute. In positions 19-71, the parameters and additional comments are to be placed. All parameters are separated with a comma. When a space is encountered, the remainder of the line is assumed to be a comment.

Alternatively, some implementations allow GPSS to be more freely constructed, but the above description also provides a clean readability.

### 2.3.2 Entities

During a GPSS simulation, certain state changes happen by the execution of actions on primitive *entities*. The actions are represented by the *building blocks*, whereas the collection of entities represent the current system state.

**Transactions**

Seeing as GPSS is a block diagram, we can state that "items" flow from one place to another, through several blocks. The flowing "items" are called "*transactions*" in GPSS. Such a transaction contains a set of parameters, which can be user-defined, or set by the system. Every transaction has a *priority* (`PRI`) and a *mark time* (`M1`). The priority identifies the importance of the transaction (the higher, the more important) and the mark time is set to the creation time of the transaction. These parameters can be altered by the `ASSIGN` and `MARK`

blocks. During execution, at most one transaction can be active at any given time, and it is the task of the simulator to update the state of the transaction w.r.t. the current state of the model.

**Standard Numerical Attributes**
Besides the creation of certain building blocks, GPSS also allows the construction of specific constructs. These constructs are called *non-mobile entities* and provide a global data structure of functionality. The most explicit usage of these entities are *standard numerical attributes* (SNAs). They can be seen as a global reference to a specific value. E.g. `N5` represents the number of transactions that have entered block 5 and `XH$var` contains the contents of *halfword savevalue* with name `var`.

Often, the `FUNCTION` statement is used in combination to a random number generator (RNG) SNA (`RN1`, `RN2`, ..., `RN8`)[6] to provide the inverse cumulative distribution function (CDF) for a randomized distribution. For more information on RNG, see *section 4.1.2*.

### 2.3.3 Chains

With transactions flowing through our system, we require a way to coordinate this behaviour. GPSS has a set of so-called *chains* on which transactions may find themselves. Each transaction must be on exactly one of:

- The *future events chain* (FEC), which contains all transactions that are due to move at some point in the future. They are kept in chronological order of departure time. Transactions that need to depart at the same time are kept in order of arrival.
- The *current events chain* (CEC), which consists of all transactions that should have moved, but cannot due to some blocking condition. The transactions are sorted by priority and in order of arrival
- A *user chain*, which is a chain that is defined by the user. Moreover in *section 2.3.8*.
- The *interrupt chain*, which is the chain that holds all transactions that are due to have control of a facility (see *section 2.3.7*), but were interrupted by another transaction.
- The *match chain*, which will not be discussed in our GPSS subset.

Additionally, whenever a transaction is on the CEC, it may also be simultaneously on a *delay chain*. Such a chain indicates the specific condition for a

[6] These generators generate values in $(0, 1000)$, unless used in combination with the `FUNCTION` statement, where they generate values in $(0, 1)$.

transaction to be blocked. Delay chains are associated with resource allocation (see *section 2.3.7*).

### 2.3.4 Scanning Algorithm

The beating heart of GPSS is its *scanning algorithm*. It defines which transactions are allowed to move when and how. It can be summarized as follows:

1. All transactions in the system are sorted by priority on the CEC.
2. The first unblocked transaction on the CEC is marked as "active" and will travel as far as possible (within the current clock time) through a sequence of blocks. Whenever a blocking condition prevents it from moving, it is copied to a delay chain. A transaction that is scheduled to move at some point in the future (due to a `GENERATE` or an `ADVANCE` block) is moved to the FEC.
3. *Step 2* is repeated until there are no unblocked transactions to be found on the CEC. When a transaction leaves a FEC, it is placed on the CEC and we return to *step 2*, as long as the termination counter is not zero.

Hence, only a single transaction moves at a certain point in time. Note that this may cause undefined behaviour when two transactions of the same priority are required to move at the same point in time. The modeler is encouraged to explicitly set priorities on transactions to prevent such issues. Furthermore, each transaction should be made unique to ensure a valid execution of the scanning algorithm in practice.

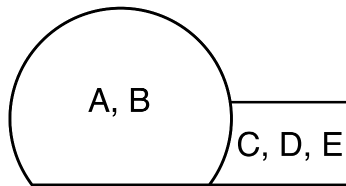
### 2.3.5 Flow

A transaction flows from a *source* (a `GENERATE` block, see *block 2.2*) to a *sink* (a `TERMINATE` block, see *block 2.3*). The `GENERATE` block generates the transactions, depending on a set of parameters. Let's ignore `A`, `B` and `C` for now. `D` indicates the amount of messages to be generated and `E` identifies the priority of the generated transactions.

The `TERMINATE` block removes transactions from the simulation. Next, it reduces the *termination counter* of the simulation with `A` units. Whenever this counter reaches 0, the simulation finishes.

In the graphical representation, we identify the flow in the model with arrows. In the textual notation, the normal block output flows into the input of the block on the following line. For instance, the following code creates 5 transactions with priority 3 that flow into the terminate block.

```
GENERATE  10,0,,5,3
TERMINATE 1
```

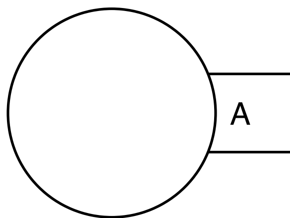



(a) Visual Notation

```
GENERATE A,B,C,D,E
```

(b) Textual Notation

Block 2.2: GPSS `GENERATE` block.




(a) Visual Notation

```
TERMINATE A
```

(b) Textual Notation

Block 2.3: GPSS `TERMINATE` block.

Obviously, a system like this is not too impressive. The `TRANSFER` block (see *block 2.4*) allows the splitting and joining of multiple flows, causing more complex systems. `A` identifies the mode of this block and the meaning of `B`, `C` and `D` is dependent on which mode is chosen. Within our subset, we'll only consider the *unconditional* (`A` is blank), *conditional* (`A` = `BOTH`) and *all* (`A` = `ALL`) modes.

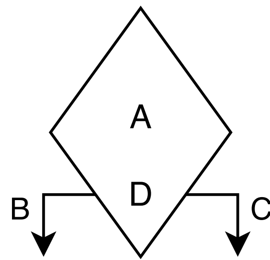
Possibly, you'd like to change certain parameters of a transaction, depending on the branch that is taken. This can be done with the `ASSIGN` block, as depicted in *block 2.5*.

Finally, a `TEST` block (see *block 2.6*) can be used to test the values of two SNAs, `A` and `B`. Within our subset, we'll assume `C` is always set to the fallback location if the condition `X` evaluates to be `false`.

### 2.3.6 Time

In GPSS, there is no definition of a specific time-unit. Instead, it is up to the user to create and analyze a model with respect to the correct time unit.

Let's revisit the `GENERATE` block with this in mind. `A` and `B` identify the time delay between transactions. That is, when a transaction is created, the next transaction will be outputted `A` ± `B` time later, where `A` equals a mean

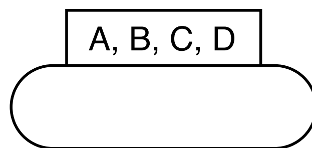



(a) Visual Notation

```
TRANSFER A,B,C,D
```

(b) Textual Notation

Block 2.4: GPSS TRANSFER block.




(a) Visual Notation

```
ASSIGN A,B,C,D
```

(b) Textual Notation

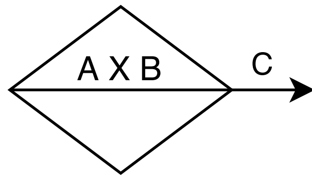
Block 2.5: GPSS ASSIGN block.

time and B the spread around this value. In fact, this corresponds to creating a random integer in [A − B, A + B]. If B is a function SNA, A will be multiplied with the result of B. This way, more complex distributions can be used. C is some offset time for the first transaction.

There is one more block that can influence the time in GPSS, which is the ADVANCE block (see *block 2.7*). It determines a randomized delay (in the same manner as the GENERATE block) and delays an incoming transaction for that amount of time. Here you can sort of recognize the ADVANCE block from our BBL (see *section 4.2.2*, *block 4.11*).

### 2.3.7 Resources

*Resource allocation* is a major aspect in GPSS. It allows for more complex and flexible systems. Let us identify a *critical section* in our models as the submodel in which only a limited amount of transactions can gain access to. Henceforth, we have an amount of *resources* that can be claimed by transactions. If there are enough resources left, access is granted. If there aren't, the transaction that requested access is blocked until enough resources become available once again. We identify three kinds of resources: *facilities*, *storages* and *logic switches*.

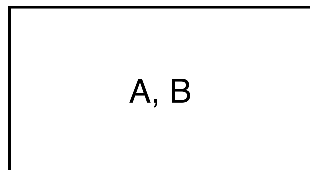



(a) Visual Notation

TEST X A,B,C

(b) Textual Notation

Block 2.6: GPSS TEST block.




(a) Visual Notation

ADVANCE A,B

(b) Textual Notation

Block 2.7: GPSS ADVANCE block.

Up to this point, all transactions were able to flow directly through the model without any issues. Unfortunately, this is thwarted by resource allocation. Whenever a transaction is blocked, it is moved to a delay chain, indicating that it waits at the block that caused a blocking condition. Multiple transactions can therefore be waiting in a single block. If the `GENERATE` block is followed immediately by a block that caused a blocking condition, the `GENERATE` block is halted until further notice.

**Facilities**

Facilities are single-access resources, meaning that only a single transaction can gain access to the resource (and therefore enter the critical section) at the same time. To access a facility, a transaction must enter a `SEIZE` block. Access can be relinquished using a `RELEASE` block. Both the `SEIZE` and the `RELEASE` blocks are represented in *block 2.8*.

Argument `A` identifies the name of the facility that is requested or released. A transaction that enters a `RELEASE` block must have previously gained access to the facility. Note that it's up to the modeler to release a facility eventually[7].

Alternatively, access to a facility can be obtained via a `PREEMPT` block. In

[7]Notice the similarity to pointers.

(a) Visual Notation of SEIZE

```
SEIZE A
```

(b) Textual Notation of SEIZE

(c) Visual Notation of RELEASE

```
RELEASE A
```

(d) Textual Notation of RELEASE

Block 2.8: GPSS SEIZE and RELEASE blocks.

GPSS, this block is one of the most complex building blocks. We will be using a simplification of this block that only contains two arguments: A (the facility) and B (the mode).

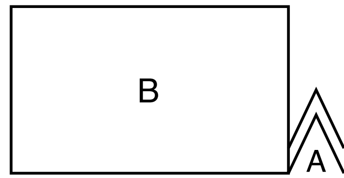
When B is not set (i.e. *interrupt* mode) and a transaction enters this block, requesting access to a facility A, it will interrupt the transaction that has obtained access through a SEIZE block. If the facility is already being controlled by a preemptive transaction, all new transactions will be blocked. If B equals PR (i.e. *priority* mode), all incoming transactions may interrupt the transactions that have a lower priority. Such an interruption moves the transaction that used to have access to the interrupt chain. Notice how this can only happen when the first transaction is on the FEC.

A RETURN block simply releases access to the resource back to the last transaction that had access. The PREEMPT and RETURN blocks are shown in *block 2.9*. [Gor75] states that, when access is released or returned, the scanning algorithm needs to restart immediately.

### Storages

Storages are resource pools of a predefined size. The size is determined with the STORAGE statement (where NAME identifies the storage name and A the capacity):

```
NAME  STORAGE     A
```




(a) Visual Notation of `PREEMPT`

```
PREEMPT A,B
```

(b) Textual Notation of `PREEMPT`

(c) Visual Notation of `RETURN`

```
RETURN A
```

(d) Textual Notation of `RETURN`

Block 2.9: GPSS `PREEMPT` and `RETURN` blocks.

These resource pools allow transactions to occupy (using an `ENTER` block) or surrender some of that capacity (using a `LEAVE` block). Both blocks are represented in *block 2.10*. `A` is the name of the storage to gain access to and `B` identifies the capacity that is requested or relinquished.

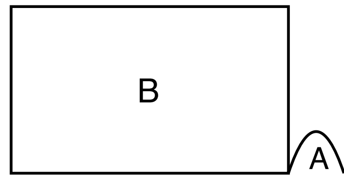
Furthermore, the space that was taken up using the `ENTER` block does not have to be the space that is freed via a `LEAVE` block. Additionally, a transaction that enters a storage does not need to be the one that leaves the storage, contrary to facilities, where the leaving transaction must have entered beforehand[8].

### Logic Switches

Logic switches are resources that can either be *set* (`true`) or *reset* (`false`). By default, all logic switches are in *reset* mode. A `LOGIC` block updates the state of logic switch `A`.

GPSS also has a `GATE` block that is able to test certain states of the resources. While [Gor75] identifies four types of `GATE` blocks, we will only be discussing the block w.r.t. logic switches within the subset under study. Both the `LOGIC` block and the corresponding `GATE` block are given in *block 2.11*.

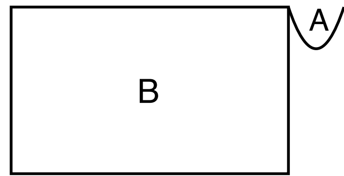
[8] This also identifies the most apparent difference between facilities and storages.




(a) Visual Notation of `ENTER`

```
ENTER A,B
```

(b) Textual Notation of `ENTER`




(c) Visual Notation of `LEAVE`

```
LEAVE A,B
```

(d) Textual Notation of `LEAVE`

Block 2.10: GPSS `ENTER` and `LEAVE` blocks.

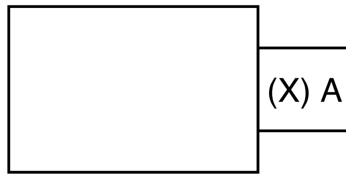
### 2.3.8 User Chains

Besides the CEC, FEC, and delay chains, GPSS also allows for custom user chains. A single process flow can put transactions on a chain using a `LINK` block and another process can take a transaction out of the chain using an `UNLINK` block (both blocks are shown in *block 2.12*). While user chains might be seen as lists, their data structure is slightly more complex.

A `LINK` block can add a transaction onto a chain that's identified by argument `A`. The order in which the transaction is added is identified by argument `B`, which is unset if it must be added to the front. When `B` equals `BACK`, it needs to be added to the back. Finally, `B` may also identify a parameter of the transaction, in which case the transactions need to be added in ascending order of that parameter. The `UNLINK` block may remove any number (`C`) of transactions from the chain. Again, this can be from the front, from the `BACK`, or by matching a transaction parameter, as is identified in field `D`. An `UNLINK` block can have three outputs:

- The normal output, to which all unlinking transactions[9] need to be sent.
- The location to which the unlinked transaction[10] is to be send (`B`). Usually, if the `LINK` block has an argument `C` (i.e. a location where to send

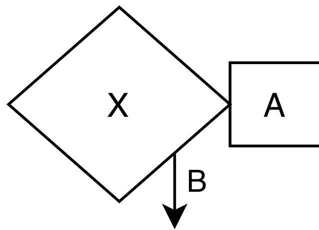
[9] The transactions that enter the `UNLINK` block.
[10] The transaction that is removed from the user chain.




(a) Visual Notation of `LOGIC`

```
LOGIC X A
```

(b) Textual Notation of `LOGIC`




(c) Visual Notation of `GATE`

```
GATE X A,B
```

(d) Textual Notation of `GATE`

Block 2.11: GPSS `LOGIC` and `GATE` blocks.

the transactions if the chain is empty), it represents the same location as the `UNLINK`'s `B` parameter identifies.

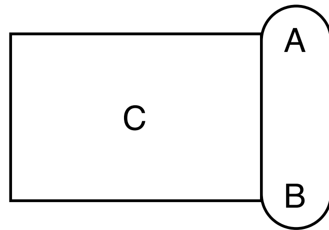
- An optional location (`F`) to send the unlinking transaction if there are no transactions on the chain and therefore the `UNLINK` failed.

### 2.3.9 Gathering Statistics

More often than not, we might be interested in certain aspects of our models. Be it the transit time, the average waiting time, the rate of transactions at a certain point, or the amount of transactions that passed through a certain block.

Some statistics are gathered automatically, like the amount of items that entered a specific block. In the case of static entities (like resources, user chains...), *table 2.2* provides an overview of the information that is gathered in those systems. Note that, depending on the entity, the interpretation of the statistics may vary. As you can see, besides resources, there are also other entities that may gather statistics.

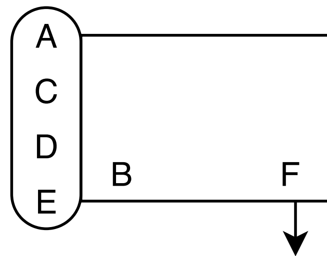
[11] Depending on the entity, it may have another meaning.
[12] Transactions that had a transit time of zero.




(a) Visual Notation of `LINK`

```
LINK A,B,C
```

(b) Textual Notation of `LINK`




(c) Visual Notation of `UNLINK`

```
UNLINK A,B,C,D,E,F
```

(d) Textual Notation of `UNLINK`

Block 2.12: GPSS `LINK` and `UNLINK` blocks.

**Tabulation**

When we're interested in the transit time of a path in our model, we might want to make use of the `TABLE` statement and the corresponding `TABULATE` block. The `TABLE` statement creates a table that, besides the mean and standard deviation of all gathered values, consists of multiple "*buckets*". These buckets represent a range in which the obtained values fall. Possibly, the obtained values need to be weighted, because they represent a larger (variable) amount of objects. To illustrate this, let's take a look at an example from [Gor75].

**Example 2.3.1** (Job Lots).

*Suppose that a transaction represents a job lot, where the number of parts in the job lot is placed in halfword parameter 5 (`PH5`). The model represents a system that processes parts by the job lot and the time to measure a job lot can be measured with the transit time. It may be necessary to recognize the different lot sizes in order to tabulate the transit time by part.*

*Let table 2.3 identify the input the table receives. Table 2.3 shows the gathered statistics of the table for the corresponding input.*

| Statistic | Static Entity | | | | | |
|---|---|---|---|---|---|---|
| | **Facility** | **Storage** | **Logic Switch** | **User Chain** | **Queue** | **Table** |
| Utilization | X | X | | | | |
| Transit Time | X | | | X | X | |
| Current / Contents[11] | X | | | X | X | X |
| Capacity | | X | | | | |
| Remaining Capacity | | X | | | | |
| Maximum | | X | | X | X | |
| Average Content | | X | | X | X | |
| Number of Entries | X | X | X | X | X | X |
| Number of Zero Entries[12] | | | | | X | |
| Status | | | X | | | |
| Mean | | | | | | X |
| Standard Deviation | | | | | | X |

Table 2.2: Overview of the gathered statistics in GPSS by static entities.

| Transaction | Transit Time | Lot Size |
|---|---|---|
| 1 | 250 | 5 |
| 2 | 320 | 8 |
| 3 | 30 | 2 |
| 4 | 150 | 4 |
| 5 | 220 | 6 |

Table 2.3: Input for *example 2.3.1*.

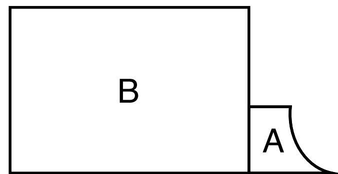
| | Nonweighted | Weighted |
|---|---|---|
| Number of Entries | 5 | 25 |
| Mean | 194.0 | 231.6 |
| Standard Deviation | 110.1 | 84.9 |
| 100 | 1 | 2 |
| 200 | 1 | 4 |
| 300 | 2 | 11 |
| 400 | 1 | 8 |

Table 2.4: Gathered Statistics for *example 2.3.1*.

*Example 2.3.1* can be written in GPSS as follows:

```
           MARK
           ...
           TABULATE   TRTM,PH5
           ...
     TRTM  TABLE      M1,100,100,A6
```

The TABULATE block (as is shown in *block 2.13*) allows us to obtain the inputs, based on a parameter of the incoming transactions. When the mark time (M1) is used, this value is subtracted from the current time and its result is tabulated. If you only want to obtain the information about a small part of the model (and not from GENERATE to TABULATE), you may indicate the starting position with a MARK block (see also *block 2.14*), which resets the M1 parameter value[13] of the transaction to that time. Obviously, the TABULATE block may also track other statistics.

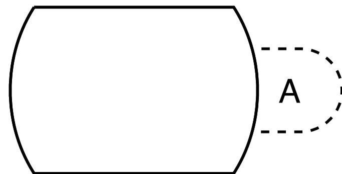



(a) Visual Notation

```
TABULATE A,B
```

(b) Textual Notation

Block 2.13: GPSS TABULATE block.




(a) Visual Notation

```
MARK A
```

(b) Textual Notation

Block 2.14: GPSS MARK block.

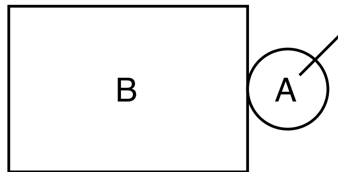
[13]Or any parameter value that is required.

Alternatively, we might be interested in identifying a bottleneck in our system and henceforth, we use the `TABLE` in *rate* mode. Here, the statistic is the amount of items that entered a `TABULATE` block over a certain time period.

**Queues**

When blocks deny access to a resource (as seen in *section 2.3.7*), the transactions will wait in the block that denied the access. Because of this, implicit FIFO queues can start to form within the model.

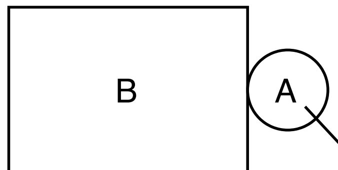
By default, exact information about these queues is not retained, but sometimes this information is quite useful. If we want to obtain the exact information about such a queue, we can surround the block in which the transactions are waiting by a `QUEUE` and a `DEPART` block. This makes the queue explicit and allows us to reason about what happens inside. The `QUEUE` and `DEPART` blocks are shown in *block 2.15*.




(a) Visual Notation of `QUEUE`

```
QUEUE A,B
```

(b) Textual Notation of `QUEUE`




(c) Visual Notation of `DEPART`

```
DEPART A,B
```

(d) Textual Notation of `DEPART`

Block 2.15: GPSS `QUEUE` and `DEPART` blocks.

Notice that the `QUEUE` and a `DEPART` blocks do not need to surround a single block, but in fact may surround a submodel instead. Therefore, these queues may be nested[14].

[14] And a transaction can therefore be in multiple explicit queues at the same time.

# CHAPTER 3

# Tools, Frameworks and Libraries

Before we dive into the creation of the building block library, it is important to get a solid understanding of the libraries and software that already exists. *Section 3.1* provides an overview of five tools that will help us get a solid understanding of the requirements for production system models. They define which functionalities are useful in a practical context, preventing us from the need to create a building block for every single function that comes to mind. *Section 3.2* will discuss some already existing DEVS frameworks, and more specifically, the libraries of building blocks they provide. Here, we will get a solid understanding on what is already out there and what we can compare our library against. At last, *section 3.3* will discuss some software that can be used to model in GPSS.

## 3.1 Tools

In order to make sure the building blocks we create are useful in practice, we'll look at existing tools and the functionality they provide. In this section, we will see an overview of five tools that will help us get a solid understanding of the requirements for production system models. *Chapter 4* will delve deeper into the actual building blocks and their functionality by referring back to these tools. For the purposes of our library, we will make sure most features that these tools provide will also be available in our set of building blocks. Yet, we still have to keep the design of DEVS in the back of our head. This inherently implies that not all functionality can be represented.

### 3.1.1 ExtendSim

In 1987, Imagine That, Inc. released the software Extend for Macintosh. It originally only allowed for continuous modeling, but in the subsequent years *discrete-events*, *hierarchy*, *interactivity* and many more features were added[1]. In 2007, they changed the name of the software from Extend to ExtendSim, which is what it is known as today.

ExtendSim is used by Stanford University, Pitney Bowes, P&G, Northrop Grumman, the Canadian army, Boeing and many more. It has a clean and logical user interface that optimizes user experience.

Most of the knowledge on this tool will be inspired upon the manual for Extend v5 [Dia+00], but all provided functionality is still available to this day, which is why all citations for this tool will fall under [Ima87].

#### Building Blocks and Functionality

ExtendSim provides about one hundred building blocks by default, separated in unique libraries that flawlessly describe their overarching functionalities. Additionally, it is possible to add more libraries (and thus more blocks) and create custom blocks. This makes the tool flexible, expandable and incredibly useful to work with. On top of that, ExtendSim comes with a debugger that allows users to easily find issues in their models.

#### Does ExtendSim map to DEVS?

It is hard to fully create an unambiguous and straightforward mapping between an ExtendSim building block and an Atomic or Coupled DEVS model. Most of the blocks themselves map quite flawlessly, i.e. they optionally take inputs, analyze and/or transform them and output the (new) values. Yet, not all blocks follow this structure.

If we look at the blocks in the *Report Library* (among others), we can clearly see that these blocks provide an analysis over the entire model or a submodel thereof. They do not take any input ports, but instead use an internal knowledge of all blocks in the model. While they might not be represented by blocks in a DEVS library, we can still provide the required functionality by making use of some simulation tracers[2]. Alternatively, in some cases, adding a collector (see *section 4.3*) might provide the required functionality.

On top of that, the ports of these building blocks are either push or pull ports (see also *section 2.2.6*). It has been described before that DEVS uses push ports and pull ports can be transformed as such.

Long story sort: most ExtendSim functionality maps onto DEVS, either by specified building blocks, (additional) connections and/or simulation tracers and manipulators.

[1] Including a release for Windows in 1995.

[2] These are possible in Python(P)DEVS.

### 3.1.2 SIMUL8

Whereas ExtendSim's main purpose lies within assembly lines and factories, SIMUL8 [SIM94] can be used to model full business processes. It's a tool that's be used for planning, design, optimization and reengineering of such processes. This can be done using their own building blocks, Statecharts[3], Buisiness Process Model and Notation (BPMN) and Value Stream Mapping (i.e. Lean Manufacturing).

To quote their own website:

> "*Launched in 1994, SIMUL8 was quickly embraced by organizations across a wider range of industries beyond manufacturing - from healthcare, to call centers and even government bodies.*"

SIMUL8 is used by Nokia, Coca-Cola, McDonalds, Cisco, Unilever, Dell...

**Building Blocks and Functionality**

What SIMUL8 lacks in a building block library (a mere 25 blocks) is made up by their numerous menus and tools.

The building blocks contain the basics of any kind of simulation on top of the most common blocks for BPMN, Work Item State Charts and Lean's Value Stream Mapping. Surprisingly, this is enough for a multitude of different models.

On top of that, SIMUL8 has (what they call) "*Ribbon Tabs*". These are some tabs on top of the screen that allow for analysis, logic and configuration of certain blocks, the simulation and the overall system.

**Does SIMUL8 map to DEVS?**

Yes, it does. If we ignore the *Kaizen* and *Manual Information* blocks (which are technically only graphical elements as far as SIMUL8 is concerned), each block takes inputs, processes them and outputs them. The *Routing Arrows* are simple connections if we assume all ports are push ports. As far as the provided functionality is concerned, most of them can be solved by adding a tracer to the simulation. The others may be implemented as building blocks.

### 3.1.3 FlexSim

FlexSim [Fle93] is a 3D simulation tool that allows users to build their own factories, assembly lines, business processes... in an actual 3D world. It was originally released in 1993 by F&H Simulations. In 1998, F&H Holland developed a first generation simulation engine for 3D objects. When F&H Holland was acquired in 2000, the development for a new tool FlexSim started, with its first release in 2003.

[3]While called "State Charts", the blocks correspond more to a glorified version of FSA.

The tool allows for analysis, visualization and improvement for real-world processes. It is used by Ford, Amazon, Michelin, FedEx, Apple, ABInBev and many, many more. On top of that, FlexSim is currently aiming towards *Industry 4.0* and, in fact, it even supports *VR/AR*, *autonomous systems* and *digital twins*.

**Building Blocks and Functionality**

Anyone that has ever worked with 3D modeling and animation engines, like Blender [Ble98], Maya [Aut98], 3DS Max [Aut96], Cinema 4D [Max90], Unity [Uni05], Unreal Engine [Epi98]... will immediately understand the controls and functions of FlexSim. Yet, because of the easy-to-use interfaces, extensive user manual and numerous tutorials, 3D experience is no necessity and anyone can get started with this tool.

There are about seventy building blocks that come with FlexSim (excluding the ones available for *process flows*), most have an accompanied 3D model, making it perfect for you to create a digital model of any system under study. If your machines don't look like the ones provided, you can also use your own 3D models.

**Does FlexSim map to DEVS?**

Before this question will be answered, it is important to note that FlexSim is 3D-oriented software, whereas DEVS is a basic concept and ideology. Therefore, any building block that is purely for visual purposes can be ignored from the question. For instance, the *Visual* section of building blocks, among others, can be excluded.

With that out of the way, there are two major remaining sections in the list of building blocks. For starters, the industrial blocks do map onto DEVS. They take a set of inputs, process them (possibly with a delay) and outputs them via one of the output ports. An issue there is here is the flexibility of these ports. Any block can have any number of inputs and outputs, all of whom do exactly the same thing. It's up to some internal delay or condition to determine block specific results. This includes routing to particular output ports.

A main caveat in this comparison is the fact that the connections and building blocks can "back up", meaning they can start overflowing when to much items arrive. We may ignore this issue if we make use of the pull/push structure, described in *section 2.2.6* and when we introduce a building block that can store all the overflowing information (see the QUEUE block, *section 4.2*, *block 4.8b*).

Secondly, there are the blocks that handle *artificial intelligence* (AI) within the model. FlexSim offers a way to also program certain paths agents[4] can take. It brings life to your models, now you can actually see the staff working and

[4] people (customers, staff...) or vehicles

transporting items outside of some conveyors, machines, pipes... In [KK00], Jae-Hyun Kim and Tag Gon Kim define MDEVS, which is an extension of the DEVS formalism for mobile agents. Hence, it is possible to extend DEVS to work with user agents, yet, we will ignore it for the purposes of this thesis.

### 3.1.4 Enterprise Dynamics

Introduced in 1998, Enterprise Dynamics (ED) [INC97] was developed by S&F Simulations[5] as Taylor Enterprise Dynamics. In 2000, when F&H Simulations was acquired, the company changed their name to INCONTROL Simulation Solutions, separating from FlexSim Software Products.

ED is a discrete-event based simulation software platform that is applied in numerous fields. It is used by Philips, KNAPP AG, Volvo, Schiphol Amsterdam Airport, Integrate, JDE...

#### Building Blocks and Functionality

The tool is object-oriented and models follow a clean workflow. Similar to FlexSim, any block (in ED called "*atoms*"), there are multiple input and output ports possible. In total, there are roughly sixty atoms available to suffice your modeling need.

Given the history between FlexSim and ED, it is difficult not to see ED as the (more theoretical and abstract) little sister of FlexSim. They both provide the same modeling functionalities and even the 3D-view ED provides, while by far inferior[6], can be made to show the same information.

#### Does ED map to DEVS?

If we use the same logic as we did for [Fle93] and ignore all AI-oriented building blocks, we can clearly see that the remaining functionality is somewhat mappable to DEVS.

Atoms are connected via channels that can be open or closed. It is not possible for items to flow over closed channels, making the model revert to an error state when such an event occurs. DEVS does not have error states by default, yet they can be seen as an event that fires, which can be captured elsewhere.

### 3.1.5 LEGO Mindstorms

The odd one out, LEGO Mindstorms [LEG13] was conceptualized in the 1980's from a collaboration between the Massachusetts Institute of Technology (MIT) and the LEGO Group as a way to do research and learn. In 1998, LEGO released the *Robotics Invention System* as LEGO Mindstorms.

[5] Yes, the very same that created FlexSim's precursor.
[6] ED is not a 3D-tool, in comparison to FlexSim, so this is to be expected.

Besides it being a reconfigurable robot made out of LEGO building blocks, LEGO Mindstorms has a programmable core that can understand any number of languages, yet they also provide some sort of Scratch [Res+09] interface for programming. It is this programming interface that we will discuss as a tool.

**Building Blocks and Functionality**

Most of the building blocks are technically the same: they send and receive signals to and from external actors or sensors. The other blocks allow for simple routing, visuals and mathematics. There is not too much to it, yet their flexibility for each block is desirable.

LEGO Mindstorms comes with a *Random* block, which has only two kinds of distributions: a *uniform integer distribution* and a *Bernoulli distribution*. Most other tools provide a more expansive set of distribution functions. See also *section 4.1.2*.

**Does LEGO Mindstorms map to DEVS?**

From the provided blocks, it is quite clear that the tool works like a Flowchart [Eur+66] editor, which is translated to machine instructions behind the scenes. Luckily, Flowcharts can be easily mapped onto DEVS.

Furthermore, [LEG13] provides a way to read and write variables for later use, which is a feature that normal DEVS can obtain via using additional input ports and a well thought-out set of connections.

## 3.2 DEVS Frameworks and Libraries

DEVS is not brand new, so it stands to reason that there are already some frameworks and libraries out there that provide an implementation of DEVS. In fact, we will be using such a library (Python(P)DEVS, [Van14]) to implement our building blocks.

[VV17a] provides a comparison between different DEVS tools. The interface, features and performance is analyzed. [Ris+17] provides an in-depth comparison of performance. Hence, we will not be focusing on performance, but mainly on which building block libraries (BBLs) are already offered within the context of DEVS. Furthermore, building blocks that are provided within examples of the tools will be considered case-specific and, therefore, these blocks will not under discussion.

### 3.2.1 Python(P)DEVS

The framework on which we will be building our BBL is Python(P)DEVS [Van14]. It is a Python implementation that supports both CDEVS, PDEVS and Dynamic Structure DEVS [Bar95]. Additionally, it is efficiency-oriented

[VV14] and distributed [VV15]. A tutorial for modeling with Python(P)DEVS, following the CDEVS formalism is given in [VV17b].

A major drawback for possible users is the lack of a building block library, forcing them to write their own blocks for the use cases they're trying to model. While this provides flexibility, often, blocks need to be remade within the context of a new system. Hence, this thesis fills this gap.

### 3.2.2 DEVSimPy

Built on Python(P)DEVS, DEVSimPy [Cap19; Cap+11] aims to provide a GUI for the modeling and simulation of DEVS. It comes with 8 building blocks and users may add more (presumably starting from the `PowerSystem` blocks they refer to in their tutorials). In *table 3.1*, the basic building blocks are listed, together with the category they belong to. Note that these are only the building blocks that work right off the bat.

| Category | Block | Function |
|---|---|---|
| Generators | FileGenerator | Extracts data from a standard CSV file and uses it with the DEVS formalism. |
| | RandomGenerator | Outputs a random integer in a range. |
| | XMLGenerator | Extracts data from a standard XML file and uses it with the DEVS formalism. |
| Collectors | MessagesCollector | Writes messages to a file. |
| | Plotly_For_Class | Plot classification results as a graph for plot.ly |
| | QuickScope | Plots the data results in a window. |
| | To_Disk | Writes results to a text file. |
| | To_Pusher | Pushes results to a web socket. |

Table 3.1: Default building blocks in DEVSimPy.

From here it becomes clear we need to provide the functionality to create both random numbers (see *section 4.1.2*) as well as read messages from tables and files (see *sections 4.1.1* and *4.5*). Furthermore, we require the functionality to obtain results (see *section 4.3*).

### 3.2.3 JDEVS

JDEVS [FB04] is a DEVS framework, written in Java and was (at the time of writing) last updated in 2001. Despite a simple example that comes bundled with the project, this framework does not provide any building blocks for users to start from.

### 3.2.4 adevs

A discrete-event system simulator (adevs, [Nut15]) is a C++ library for constructing models according to the PDEVS and Dynamic Structure DEVS [Bar95] formalisms. It is not a framework and should therefore be linked correctly to C++ code that uses it, but, nevertheless it is a powerful library that provides full control to the user. Because it *is* a library, the only associated building blocks it provides are example-specific and can therefore not be extended to our concept of a BBL.

### 3.2.5 DEVS++ and DEVS#

Both DEVS++ [Hwa07b] and DEVS# [Hwa07a; Hwa07c] are libraries for the DEVS formalism, written respectively in C++ and C#. With the former being the original version and the latter a port to C#, which is preferred by the author. This allows us to look at both libraries as one within the provided context.

Unfortunately, neither of these libraries come with a set of predefined building blocks and therefore they still require all users to build their own, similar to adevs and Python(P)DEVS.

### 3.2.6 DEVS-Suite

The successor of DEVSJava [SZ98], DEVS-Suite [KSE09], is a tool that allows for the visualization of Coupled model simulation, event injection and simulation tracking. Both Atomic and Coupled DEVS models need to be written in Java and compiled, before loaded in the tool. By default, no building blocks are provided with the tool. Yet, when downloading all tools as mentioned in [VV17a], the building blocks required to create the sample models are included.

### 3.2.7 DesignDEVS

Created by Autodesk as a thought-experiment, DesignDEVS [GBK16] is an idea for a software application, specifically designed to help researchers collaborate on simulation projects. The main goal of the concept was to create a way to represent DEVS in such a way that it became easily understandable for non-programmers [Mal+15].

It would have allowed for visual coupling of complex models and the creation of new Atomic DEVS models, using Lua. Important to note is that it would not fully follow the DEVS formalism, i.e. the $\delta_{int}$ and $\lambda$ functions were merged into a single function that is allowed to change its state and generate output. This was done for efficiency purposes[7]. Autodesk never disclosed any ideas for a building block library.

[7] Yet, its correspondence to CDEVS or PDEVS can be easily proven.

### 3.2.8 DEVS-Ruby

DEVS-Ruby [Fra+14] is a DEVS simulator, written in Ruby. Currently, it is replaced by Quartz [Fra+18], written in Crystal. Whereas the original simulator had some common building blocks available (see *table 3.2*), Quartz does not.

| **Category** | **Block** | **Function** |
|---|---|---|
| Generators | CosinusGenerator | Generates values according to a cosine function. |
| | SinusGenerator | Generates values according to a sine function. |
| | CSVRowGenerator | Reads a standard CSV file row by row and outputs its data. |
| | RandomGenerator | Generates random integers in a range. |
| | SequenceGenerator | Generates a sequence of integers in a range. |
| Collectors | AsyncTempfileCollector | Writes a trace to a file. |
| | CSVCollector | Writes messages to a file, following the CSV format. |
| | DatasetCollector | Writes messages to a dataset. |
| | HashCollector | Collects the messages in a *hashmap* (time, message). |
| | PlotCollector | Creates a Gnuplot [WK04] from the arriving messages. |
| | TempfileCollector | Writes messages to a temporary file. |

Table 3.2: Default building blocks in DEVS-Ruby.

As you can see, we can make the same conclusions as we did in *section 3.2.2*. We need at least the generation of events (*section 4.1*) and collection thereof (*section 4.3*).

## 3.3 GPSS Tools

Despite being somewhat out of fashion, there are still a few tools that still provide a possibility to model in GPSS. Note that the focus lies on the tools that are still readily available and can be downloaded, at the time of writing.

Whereas [Stå+11] also mentions GPSS/H, STX and Proof Animation as valid GPSS tools, they were created by the Wolverine Software Corporation, which seems to have gone out of business[8]. Similarly, GPSS V and GPSS/360 [Gou69] are not supported by IBM anymore.

In *appendix C*, there is an overview of which GPSS blocks are available in which tools. This table is an expansion of *table 1* from [CC09].

### 3.3.1 HGPSS

In [Cla92], the GPSS formalism has been extended to HGPSS, which introduces hierarchy. Therefore, a hierarchical description that corresponds to this formalism can be simulated with the corresponding C++ code.

### 3.3.2 GPSS World

GPSS World [Min10] is a lightweight GPSS simulator, created by Minuteman Software in 2010. It implements a powerful simplification of the GPSS formalism that is introduced in [Gor75]. Despite being somewhat rough around the edges[9], it is quite user-friendly and provides an extensive documentation.

### 3.3.3 GPSS/H

Developed by Wolverine Software, GPSS/H [Cra97] is a *command line interface* (CLI) for modeling in GPSS[10]. It allows you to enter a filename, which contains a full GPSS description and creates a `*.LIS` file, which contains the report of the simulation. Because Wolverine Software does not seem to exist anymore, there is not much documentation on the usage of this tool.

### 3.3.4 JGPSS

JGPSS [CC09] is a Java-based framework that is intended to teach GPSS to students. It allows users to graphically create a set of processes by linking blocks together. The tool does not implement all functionality that is described in [Gor75], but nevertheless it is a useful tool for modeling in GPSS.

### 3.3.5 aGPSS

aGPSS [Stå99] claims to be the most modern version of GPSS and provides modeling in both the visual and the textual concrete syntax. There is an extensive documentation and focuses mainly on making GPSS accessible for students.

[8] Nevertheless, GPSS/H will still be discussed.

[9] Some features have been “under development” for over a decade.

[10] Technically, GPSS/H allows modeling in a superset of GPSS.

When compared to [Gor75], aGPSS makes use of a different, smaller, set of blocks. For instance, the TRANSFER block is replaced by a GOTO block that provides less possibilities. This is the reason we will not be using this tool as a baseline for GPSS.

### 3.3.6 AToM$^3$

AToM$^3$ [LV02a; LV02b] is *A Tool for Multi-formalism and Meta-Modelling* that's under development at the Modelling, Design and Simulation Lab (MSDL). It's the precursor of AToMPM [Syr+13]. Besides being quite powerful in numerous formalisms, meta-modeling [Küh06] and *model transformations* [Küh+10], it also provides a graphical user interface for modeling in GPSS. From such a graphical representation, a textual denotation is generated, which can be simulated by other tools to get the required results.

# CHAPTER 4

# PythonDEVS-BBL

As is shown in *section 3.2*, there are quite some DEVS frameworks out there, but most of them are lacking a building block library (BBL) for new users to start modeling right off the bat. Often, there is a required overhead of creating the building blocks that are necessary within the context of your problem domain. *Section 3.1* in its turn provided a general idea of building blocks that are used in professional contexts for discrete-event modeling.

This chapter will describe a set of building blocks, based upon the information gathered from the tools, frameworks and libraries that were described in *chapter 3*. These blocks will follow the formal CDEVS formalism, but may be expanded to PDEVS by the interested reader. Additionally, we want to make sure the described BBL is both *extensive* (i.e. many building blocks can be used for many problems in numerous domains), as well as *deficient* (i.e. we cannot possibly present building blocks that span the full semantic domain, so we will provide a set of blocks that can be used in many areas of this domain, omitting rare constructs and functions).

Furthermore, the BBL will be written specifically for Python(P)DEVS and can therefore exploit some subtleties that Python has to offer. Furthermore, it will make use of some initialization features, as in [VV18], by making use of some class constructor arguments. Within the pages of this paper, building blocks are identified with the Small Capitals font style.

Throughout this chapter, we will discuss Generators (4.1), Queueing Systems (4.2), Collection Techniques (4.3), Mathematical Blocks (4.4), I/O Handling (4.5), Data Transformations (4.6) and Routing (4.7).

## 4.1 Generators

Whether you're modeling a business process, a factory, a flow system, or a waiting line in a store, more often than not you will be needing a generator. Generators are small building blocks the generate either items, or numbers that can be used by the rest of the system. We identify two categories of generators for numbers: *standard generators* (see *section 4.1.1*) and *random number generators* (RNG; see *section 4.1.2*).

Alas, we do not always want to generate a number, but we might need an actual *item*. While we could introduce an *item generator*, we would lose the multitude of possibilities presented with standard generators and RNG. If you really need an item, one could transform a number to an item (moreover in *section 4.6*).

### 4.1.1 Standard Generators

*Standard generators* is the category of generators where the generated results are incredibly predictive and reusable, because items are generated according to a mapping of a time $t$ to a value $v$. This can be either a function $f(t)$, or a table with columns $t$ and $v$.

#### Functional Generators

Our function $f(t)$ can be as simple or as complex as we'd like, depending on what we need in our model. The simplest case is a *Constant* block, as it is known in [Ima87] and [LEG13], that continuously outputs a predefined value. We will define a CONSTANT GENERATOR (see *block 4.1*), where $f(t)$ continuously outputs the constant value $c$. A generator like this can be useful for a constant flow of items.

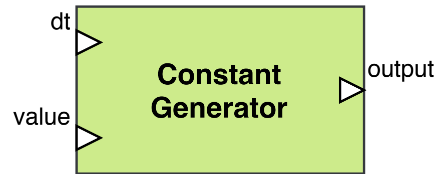
The CONSTANT GENERATOR has two inputs: *dt* and *value* that respectively allow for updating the time delay between outputs and the value that's being outputted. Note that a change in *dt* should keep the already passed time in mind. This may lead to strange behaviour when *dt* is smaller than the already passed time[1]. The output is sent periodically over the *out* output port.

A more complex, but presumable often used version of $f(t)$ is the following:

$$f(t) = \begin{cases} IC & \text{if } t = 0 \\ f(t-1) \cdot c + v & \text{otherwise} \end{cases}$$

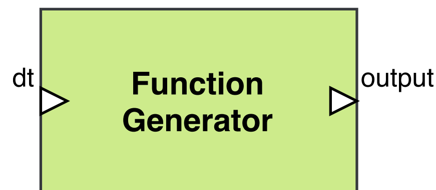
Where we have $IC$ as our initial condition for $f$, i.e. the very first value to start from, $c$ a multiplier to allow for both arithmetic and geometric progres-

[1] To prevent ambiguity, we will assume that the next output must be sent immediately.




Block 4.1: CONSTANT GENERATOR building block.

sions; and $v$ a value to increase our output with. When using this version of $f(t)$, we talk about an INCREMENTOR that basically allows for mathematical progressions. In its most rudimentary form, we have $f(t) = t$ (when $IC = 0$ and $v = c = 1$, assuming our time delay also equals 1).




Block 4.2: FUNCTION GENERATOR building block.

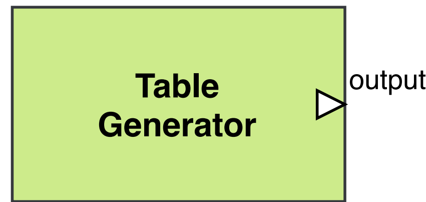
Now, of course, it does not have to end there. A FUNCTION GENERATOR (see *block 4.2*) allows the user to precisely identify which function $f(t)$ to use. This way all kinds of standard generators can be represented. The *dt* input provides a way for the user to change the delay between outputs.

We will classify the INCREMENTOR as a special case of the FUNCTION GENERATOR.

**Generating from Tables**

But what when the input data cannot be represented with a function? Simple, we can use a TABLE GENERATOR (see *block 4.3*) that has a lookup table of all values $v$, for all times $t$. Additionally, some complex functions may take too much time, or may require more memory than the hardware supports. This is usually the case for complex recursive functions. Anyhow, in those cases one might consider generating a table of all the values and use a lookup table instead, similar to sine wave (or other function) implementations in old calculators. For unknown values, different interpolation mechanisms may be used. Whenever the simulation time reaches $t$, $v$ will be outputted.

[SIM94] allows tables that generate items to repeat as soon as they have outputted all their values. For instance, if we know that we have a certain number of customers arriving for each hour, which remains consistent over several days (e.g. every day, at 8 a.m., there arrive 3 customers, at 9 a.m. there arrive 10 customers and at 10 a.m. there arrives no one), we can put this into a table and make the table repeat itself at the end.




Block 4.3: TABLE GENERATOR building block.

Henceforth, our TABLE GENERATOR also has a constructor argument indicating its repetitiveness.

### 4.1.2 Random Number Generators

To illustrate the use of randomization in models, imagine the following scenario:

**Example 4.1.1** (Sportswear Store)**.**
*You are the owner of a big sportswear store and a heatwave has been announced. From your experience, you know, due to the warm weather, more people will come by and buy some material they need to cool down. Now, you want to hire more staff to man the cope with the demand of customers, but you have no idea when best to put them to work. And then you remember this is a problem you can model.*

*You boost off, modeling your store when it hits you: you don't know how many and when customers will arrive, so you need to guess. Only you know this cannot be a constant stream, because of the realism you want in your model. It's also probably not predictable with a mathematical progression, or a function at all.*

*So you decide you need a table. The first day of the heatwave, you record when customers arrive and when they leave. But the second day comes by and soon you realize that the table you created does not match that day's customers at all. You realize it's random. Probably predictable with some sort of statistical analysis, but random nonetheless.*

Admittedly, the context of the above-mentioned problem is quite specific, but the problem is not. In modeling contexts, randomness occurs incredibly often, mainly because it also introduces realism into our models. And this randomness can be obtained via *random number generators* (RNG).

RNG are generators that have one simple job: generating a *random* number. "*Random*" being the operative word, because what is random?

Imagine throwing a fair six-sided die six times. No matter what you threw, the outcome must always classify as *random*, theoretically that is. There is no formula, no correlation between each of the throws and every result has as much

chance as any other for appearing. But what if you threw $2-2-2-2-2-2$ or $1-2-3-4-5-6$ ? The result is still random, only we do not seem to *perceive* it as being random [LEc01].

Within those sequences, we can easily see a pattern. In fact, the human mind is, albeit subconsciously, always doing pattern recognition[2]. Sometimes we see faces and shapes where there are none (pareidolia), or, as we have here, we identify patterns within randomness (patternicity, [She08]).

Would the occurrence of these sequences in a RNG make for bad randomness in a system? Probably. It depends on the context in which the user decides to use the generators. Are some generators better at overcoming this problem? Definitely! The quality of an RNG is therefore dependent on its *predictiveness*.

#### Pseudo-Randomness

Before we go into detail on different kinds of RNG, it is important we are aware of some additional concepts, which, in return, will yield a good enough set of criteria to test our RNG.

Let's say you have a perfect, flawless RNG that reduces the above-mentioned *predictiveness* massively. And now you want to conduct an experiment that uses an RNG, but in such a way our experiment always yields the same results. Even though this seems to reduce its randomness, the usefulness of the possibility to do this is massive. Many statistical tests and simulations require us to have *reproducibility* in our system. This way, if you conduct an experiment, and require its outcome to be the same, you can do so.

But we can't actually speak of "random" anymore, because it's randomization is set to be constant. This is why, when we talk about RNG, what we actually mean is *pseudo-RNG*. The most common generators are the *linear congruential generator* (LCG) and the *multiplicative congruential generator* (MCG), which is a special case of the LCG.

Another issue we have with randomness is the fact that computers are not good at it. This is mainly due to the fact that any generation of randomized data from a computer follows a static algorithm or mathematical formula, which is predefined. While the randomness can be introduced via a physical device, like noise from electrical diodes, computer-generated RNGs aren't actually made to be random, but merely *appear* to be. This is yet another reason we call them *pseudo-RNG* or *algorithmic RNG* [LS07].

#### Randomized Distributions

Within the domain of computational statistics, we might be interested in having an RNG that generates numbers according to some distribution; i.e. the

[2] This is even a branch within psychology and cognitive neuroscience! And its existence is not even a bad thing either. It allows us to draw, recognize (emot)icons and determine that the black camouflaged face in the bushes is, in fact, a panther.

outcome is distributed according to a *probability density function* (PDF). Do we create a new RNG for each PDF we want to sample from? Obviously not. So how do we solve this then? Luckily the problem of creating an RNG can be simplified:

1. Generate imitations of independent and identically distributed (i.i.d.) random variates, having a uniform distribution over the interval $(0, 1)$.

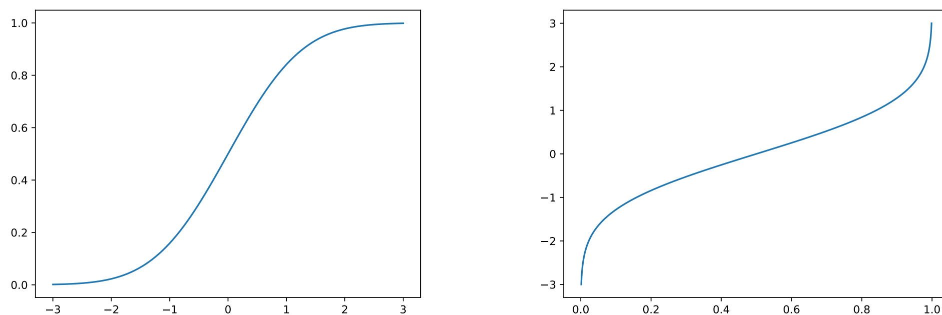
2. Apply transformations to these variates, so they match up to arbitrary distributions.

If we define the first step (generation i.i.d. $U(0, 1)$) as being the RNG itself, we just need a set of transformations that apply, no matter which RNG was used. As mentioned in [LEc12; Dev86; Law14], the simplest way of generating a random variate $X$ with cumulative distribution function (CDF) $F$ from a $U(0, 1)$ random variate $U$ is to apply the inverse of $F$ to $U$:

$$X = F^{-1}(U) \stackrel{\text{def}}{=} \min\{x | F(x) \geq U\} \tag{4.1}$$

For completeness' sake, we can prove this theorem as follows:

$$P\left[X \leq x\right] = P\left[F^{-1}(U) \leq x\right] = P\left[U \leq F(x)\right] = F(x)$$




(a) Cumulative Distribution Function, $F(x)$

(b) Inverse Cumulative Distribution Function, $F^{-1}(x)$

Figure 4.1: Plots for the Standard Normal Distribution.

Take a look at *figures 4.1a* and *4.1b*. In the former, the CDF of the standard normal distribution function is shown. The same CDF, with its axes swapped, is displayed in the second figure. Basically, what the proof says is: we generate a random value, which we use as the $x$-value of the inverse CDF of our distribution. This value will be in the range of $(0, 1)$, per definition of

the CDF. Seeing as our CDF will always be a *bijection*[3], the inverse CDF will always yield a corresponding $y$-value for each input.

If we now generate a sequence of random values in the range $(0, 1)$ and, for each of these, we return their inverse CDF $F^{-1}$, then the resulting sequence will be distributed according to the original PDF.

Thus, we've restricted our problem domain for finding good RNG to finding a good RNG that generates numbers i.i.d. $U(0, 1)$. Please be aware that both 0 and 1 are exclusive within this context, seeing that $F^{-1}(U)$ is often infinite when $U$ is 0 or 1. However, within the analysis in the RNG, one may assume 0 is admissible, because this allows for major simplifications, while having close to no difference in the mathematical structure of the generator [LEc12].

#### Inverse of the CDF

We're left with one single question in this problem: how to find the inverse of the CDF $F$, called $F^{-1}$? [Law14] discusses seven different techniques on how to do this[4], all of whom will be summarized below. We will also discuss some advantages and disadvantages for all methods. Additionally, one more method will be added, that was not mentioned in [Law14], but is the core behind the code from [LS07].

**Closed-Form Formula.** Depending on your expertise and algebraic proficiency, you might want to create the equation(s) for $F^{-1}$. And while this method will yield the best results and is the most precise, there are a few caveats.

As anyone with a lot of experience with problems like this will tell you, an error is easily made and often $F$ has pre- and postconditions. Even if they seem simple and often logical, it is important to also transform these into the pre- and postconditions of $F^{-1}$. If you don't, you might turn a blind eye for certain edge cases, which may yield invalid results, but are hidden behind the randomness of the RNG.

Also keep in mind not all CDF can be defined under a closed-form formula. Even if they can be, $F$ might become so complex, the computation thereof can be such an overhead we might want to take a different approach. In *table 4.1*, you can find a list of some commonly used distributions and their inverse CDF, under a closed-formula (based on [Law14] and [Hek16]).

**Inverse-Transform Method.** Also called *Smirnov transform* [Fau19], the *inverse-transformation* method is based on the intuition that the PDF $f$ of

[3]Which can be easily proven via the fact that a CDF is a continuous, increasing function

[4]The first of which he does not denote individually, but was added in this paper for completeness.

| **Distribution** | $F$ | $F^{-1}$ |
|---|---|---|
| Continuous Uniform | $F(x) = \frac{x-a}{b-a}$ | $F^{-1}(x) = a + (b-a) \cdot x$ |
| Exponential | $F(x) = 1 - e^{-\lambda x}$ | $F^{-1}(x) = -\frac{\ln(x)}{\lambda}$ |
| Weibull | $F(x) = 1 - e^{-(\frac{x-v}{\lambda})^k}$ | $F^{-1}(x) = v + \lambda \sqrt[k]{-\ln(x)}$ |
| Log-Logistic | $F(x) = \left(1 + (x/\alpha)^{-\beta}\right)^{-1}$ | $F^{-1} = \alpha \left(\frac{y}{1-y}\right)^{1/\beta}$ |
| Discrete Uniform | $F(x) = (\lfloor x \rfloor - a + 1) / n$ | $F^{-1}(x) = a + \lfloor (b-a+1) \cdot x \rfloor$ |
| Geometric | $F(x) = (1-p)^x$ | $F^{-1}(x) = \lfloor \log_{1-p}(x) \rfloor$ |

Table 4.1: Closed-formula form of CDFs (and their inverse) of commonly used distributions in modeling. See also *appendixes A.1* and *A.2*.

a random variable can also be interpreted as the relative chance of observing variates on different parts of the range.

Therefore, we know that if we sample $f$ in $n$ equally sized samples, where its $x$-values $\{x_0, x_1, x_2, ..., x_n\}$ cover the entirety of $f$'s domain and its $y$-values $\{y_0, y_1, y_2, ..., y_n\}$ the corresponding function-values of $f(x_i)$, the corresponding value in the CDF $F$ can be approximated as follows:

$$F(x_i) = \sum_{k=0}^{i} f(x_i) = \sum_{k=0}^{i} y_i$$

Our $F^{-1}$ can now be identified in 3 ways:

1. $F^{-1}(u)$ is the first value $x_i$ where $F(x_i)$ exceeds $u$
2. $F^{-1}(u)$ is the last value $x_i$ where $F(x_i)$ does not exceed $u$
3. $F^{-1}(u)$ is an interpolation between the last value $x_{i-1}$ where $F(x_{i-1})$ does not exceed $u$ and the first value $x_i$ where $F(x_i)$ exceeds $u$

An algorithm for the first of the options is given in *algorithm 1*, but can be easily and logically expanded to the other options. The third option is illustrated in *figure 4.2*. The blue line identifies the standard normal distribution and the orange area shows the area that's being computed.

This method is especially useful if there is no known closed formula to be found for a certain CDF. Unfortunately, this algorithm merely calculates an estimate of the value and on top of that, we need to keep in mind this algorithm needs to run on a computer. If our $f$ is precise and incredibly complex,

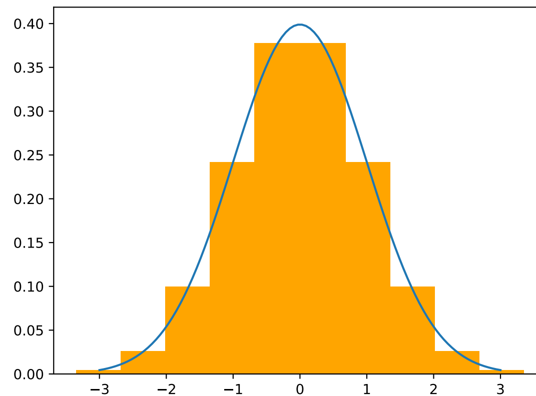



Figure 4.2: Inverse-Transformation with interpolated values.

it is perfectly possible there are rounding errors and computational errors. Additionally, assuming $f(x)$ can be computed in $O(1)$ time, the algorithm takes $O(n)$ time in a worst case scenario.

**Algorithm 1** Compute an estimate of the inverse of CDF $F$, based on PDF $f$, using inverse transform sampling.

```
 1: procedure INVERSE TRANSFORM(u)
 2:     s ← 0.001                    ▷ Step size s. The smaller the more precise.
 3:     x ← 0
 4:     p ← f(x)
 5:     while u > p do
 6:         x ← x + s
 7:         p ← p + f(x)             ▷ p contains the estimate of F(x)
 8:     end while
 9:     return x
10: end procedure
```

**Composition.** In the rare case our distribution $F$ can be expressed via a convex combination of other distributions $F_1$, $F_2$, ..., we might want to use *composition*. Of course this is only useful if the other distributions are less complex. We assume that for all $x$, $F(x)$ can be written as

$$F(x) = \sum_{j=1}^{\infty} p_j F_j(x)$$

where all $p_j$ are probabilities[5].

An algorithm that follows this formula can be summarized in one sentence: we choose a $F_j$ with probability $p_j$ that corresponds to the area you're looking at. Intuitively, you just divide your CDF into different areas that correspond to the set of distributions. Depending where the random variable in $(0, 1)$ falls, we choose the corresponding distribution for its computation.

**Convolution.** If the desired random variable $X$ can be expressed as a sum of other variables $Y_1$, $Y_2$, ... *convolution* can be used. All we have to do is generate all $Y_i$ and return their sum. It's a clean algorithm, but rarely used and incredibly inefficient (as described in [Law14]).

**Acceptance-Rejection.** When the other methods fail or are inefficient, the *acceptance-rejection* method might just work out. We require a special function $t$ (the *majorizing function*), which is an upper limit for $f$. From this $t$, we will generate a new density function $r$ which we will use to determine an approximation for $F$. Nevertheless, this method is an excellent use case when you want to simplify your distribution.

**Ratio of Uniforms.** Based on a strange property between random variables and their ratio, the inverse CDF can be computed. The interested reader is referred to [Law14] for the proof and mathematical explanation of this method. It uses two uniformly distributed variables that lie around the distribution to approximate its value.

**Special Properties.** Last but not least, it is also possible to make use of special correspondences between different distributions. For instance: if $n$ and $p$ are large enough, a binomial distribution can be approximated using a normal distribution.

**Table Method.** Last but not least, there is an incredibly simple method for finding the inverse $F^{-1}$. We sample our $F$ at equal ranges and for each sample, we store $(x, y)$ in a table. If we want to know $F^{-1}$, all we have to do is read the table the other way around. All values we do not find in our table can be interpolated[6] from the surrounding values. In order not to lose performance, our table lookup algorithm must be fast. Again, this approach is not problem-free.

No matter which interpolation method is chosen, this will always be a mere approximation of the actual value. The larger the sample size, the less precise

[5] Meaning they sum up to 1 and are all larger than or equal to zero.

[6] Either linearly, or using more complex methods.

this estimate. We also have to keep in mind that, depending on the sample size, the table can be tiny or massive. The bigger the table, the more precise $F^{-1}$, but also the more memory we require for storing this table. If it's smaller, we lose precision, but gain in memory. Additionally, many distributions have additional attributes. Sizes, probabilities for success, degrees of freedom, shapes, rates... There are a lot of different possibilities, especially if they are combined. Our table therefore can become $(m+1)$-dimensional if we have $m$ different attributes. Furthermore, most of these attributes have actually an infinite range, making it close to impossible to cover all grounds.

### Checking the Inverse

Let's say we have a valid inverse $F^{-1}$ of our CDF $F$, but how do we make sure we're right? It is not enough to state that both functions are each others inverse, seeing as there are many errors to be had, in all possible ways to compute our $F^{-1}$. Of course, this is an issue that's only of importance for statistical analysis or within the domain of Software Testing.

Either way, we can use a *goodness of fit* test for comparing distributions with known and tested datasets or software. The most common *goodness of fit* tests are the *Kolmogorov-Smirnov test*, the *Anderson-Darling test*, the *Shapiro-Wilk test*, the *Chi-squared test* and the *Normality test*.

### Which Distributions to use?

Given the tools described in *section 3.1*, we can determine a set of all the distributions that should be under consideration. A table summarizing all distributions w.r.t. these tools and [Law14] is given in *appendix A.1*.

Whenever there are at least two tools[7] that provide a certain distribution, we consider it important enough to add to our library. Additionally, the *Student's t*, the *Fisher-Snedecor* (or simply "*F*") and the *Zipf* distributions were also added to this list. In *appendix A.2*, you will find an overview of all provided distributions, their uses and how $F^{-1}$ can be obtained.

### RNG Quality Criteria

Before we go take a look at different types of RNG, let's first identify some requirements for "goodness". As mentioned before, we want it to generate a *good imitation* of a sequence of independent uniform random variables. We also want to decrease the *predictiveness* of such a sequence. Another important condition, especially within the context of *real-time events*, is for the RNG to be *efficient* (i.e. a low time and space complexity).

Some RNG need to be *repeatable* in order to better determine results and edge-cases. And we also want it to be *portable* over different software/hardware environments.

Let's say our RNG is i.i.d. $U(0,1)$ and generates the sequence $u_1, u_2, ..., u_n$

[7] Including [Law14] as a tool for these purposes.

before repeating itself. If our $n$ is too small, we get a predictive sequence by repetition, i.e. by remembering all the $u_i$ and reproducing it. In a professional casino, counting cards can get you thrown out and banned, so having RNG that make this easy, for, say, a card game, does not seem logical. This is why we want to have a *long period* within our algorithms. Finally, there is also the requirement of efficient *jump-ahead* methods that allow for a quick computation of any $u_i$ based upon a $u_j$ $(j < i)$.

In total, we have seven requirements to test for any given RNG. From these results, we can deduce the best fit for our problem. Dependent from our context, we might want other requirements to perform better. For instance, cryptology-related applications or casinos might want a higher unpredictability and have no need for a jump-ahead. Arrivals of trains in a simulation might not need such a high unpredictability, but might instead prefer repeatability to determine and identify bottlenecks.

All these requirements can be identified via statistical tests, many of which described in [LS07]. As this paper states, please be aware that no universal test or battery of tests can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. All we need is a testing framework and a set of RNG. Luckily, TestU01 [LS07] and RngStreams [LEc+02; LS03; Kar+14] provide both.

Do note that RngStreams makes use of multiple streams to ensure better randomness, which may not be preferred by some users. In order to allow a model in PythonDEVS-BBL to yield the same results as another model in Python(P)DEVS that did not use RngStreams, it is made possible in the library to choose one of three RNG stream generators[8]:

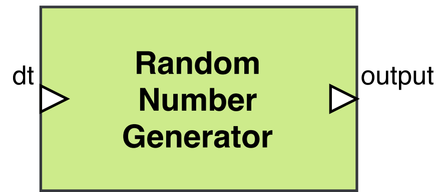
- RngStreams
- Python's builtin `random` module [Pyt]
- Numpy's `random` module [Oli+95]

Whenever RngStreams cannot be found, a warning is shown and Python's `random` module will be used instead.

#### RNG Building Block

With all this knowledge in mind, we can define a Random Number Generator building block (see *block 4.4*). The RNG is constructed with a few attributes that defined the generator: the *seed* that's used and the *distribution* to which it must map. Additionally, the block holds internally the delay until next generation, which can be altered with the *dt* input.

[8] Take a look at the documentation on how to do this.

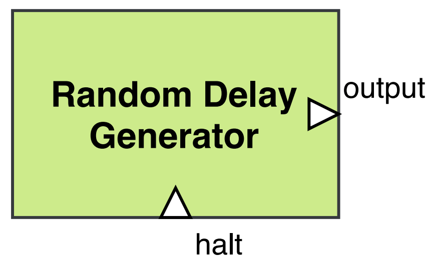



Block 4.4: Random Number Generator building block.

**Randomizing the Delay**

Whereas the Random Number Generator generates random numbers as input for a model, often it is useful to generate messages in a model with a randomized delay. The Random Delay Generator, or RDG, allows for this functionality. It is graphically represented in *block 4.5*.

The generator outputs a number $k$ over the *output* port, according to the values of the internal RNG, unless it was halted, which is indicated by the boolean value entering on the *halt* input. The value of $k$ is indicative of the amount of messages the generator has created, minus some base $B$; i.e. if $c$ represents the amount of generated messages, $k$ equals $B + c$. Additionally, we will provide a *max* attribute to the block, indicative of the amount of messages it must generate before termination.




Block 4.5: Random Delay Generator building block.

### 4.1.3 Using Stock

If you're modeling a supermarket, you probably don't have an infinite supply of items. Especially if your model is meant to determine when the shelves need to be refilled or when a truck with new products needs to arrive. The products you have available for sale are called the "*stock*". Even production lines of factories might need to use stock if they work with batches of specific items.

Whereas it would be perfectly possible to introduce a custom building block for tracking stock, it can also be created by a valid chaining of other building blocks like generators, queues (see *section 4.2*), transformers (see *section 4.6*)

and guards (see *section 4.7.6*). Because of the uniqueness of stocks within certain scenarios, no Coupled DEVS was created for this purpose. Instead, an example for such a stock is given in *figure 4.3*.

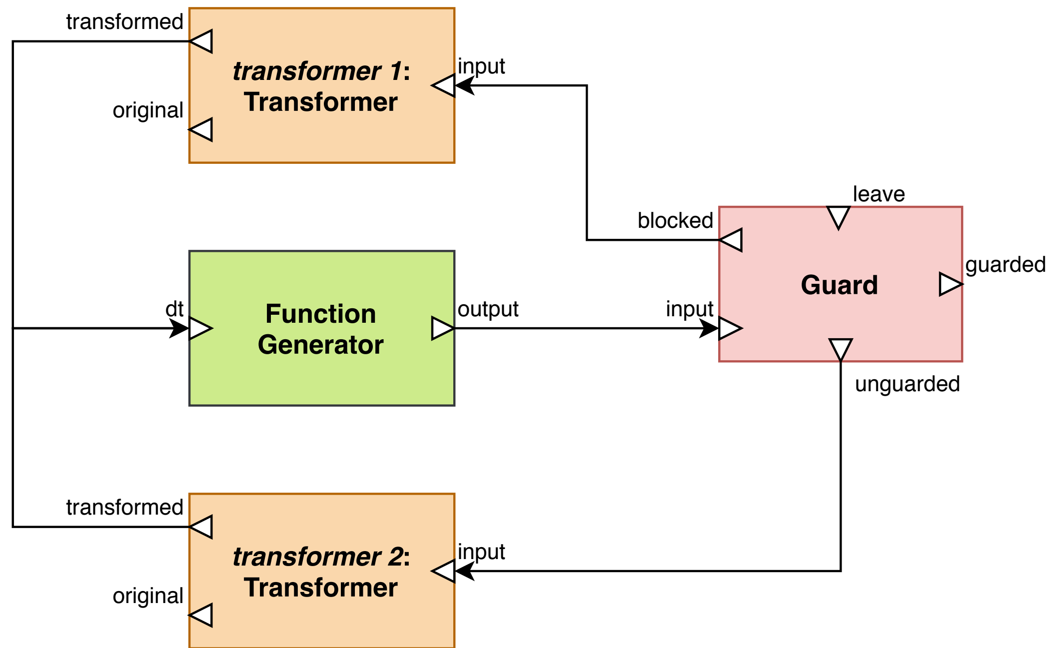



Figure 4.3: Example on how to handle stock with a combination of building blocks.

In the figure, `transformer 1` will always output infinity ($\infty$) on its *transformed* output[9], whereas `transformer 2` will output zero on its *transformed* output[10]. For both blocks, the *original* output port will be ignored. The Function Generator is used as a general example of generators.

Let's start in our Function Generator with a *dt* of 0. Immediately, there will be a set of items, determined by the Guard block that can enter the system over the *guarded* output, possibly linked to a Queue. When the maximal capacity was reached, the Function Generator will be halted. When the stock is updated (i.e. when an input arrives on the *leave* port), the Function Generator will automatically create new items until the stock is filled once more.

### 4.1.4 Firing Single Events

Whereas the generators we have discussed so far continuously fire events, it might be useful to have a block that only fires once. Similar to the Table

[9]Halts the generator.
[10]Restarts the generator.

GENERATOR (as discussed in *section 4.1.1*), but with a single record in its table. Because the TABLE GENERATOR might contain a lot of overhead for such a feature, we will provide a SINGLE FIRE block (see *block 4.6*) that does just that.

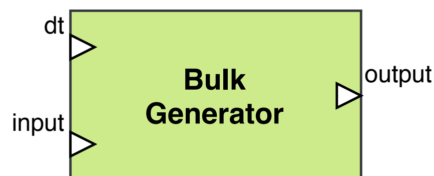
Block 4.6: SINGLE FIRE building block.

The block takes an *item* and an absolute time $t$ in the constructor. The $t$-value defines when to send the *item* over the *out* output port.

### 4.1.5 Generating in Bulk

While the currently provided generators already allow for a lot of functionality, there is a single concept that is not yet supported: creating a customizable amount of messages. In [SIM94], you can ask the *Start Point* to generate $n$ messages at the current time frame, which is incredibly useful in a context where you want to create $n$ customers every time unit.

In order for us to still support random number generation and all other methodologies, let's create a BULK GENERATOR (see *block 4.7*) that takes $n$ as an input and spews out $n$ unique customers. These customers basically correspond to the amount of items the block has outputted at that time.




Block 4.7: BULK GENERATOR building block.

Similar to the CONSTANT GENERATOR, this block has a *dt* input where the delay between the messages can be altered at runtime. Whenever there is an input when the BULK GENERATOR still has items to generate, the set of messages to generate is increased with the new size.

## 4.2 Queueing Systems

A *queue* is a container in which an ordered collection of items is stored. Items can enter this queue via a so-called *enqueue* operation and they can leave via a *dequeue*. Think for instance about the waiting line before the cash register at your local supermarket, or when you're waiting in line for an attraction at an amusement park. Colloquially, we call these waiting lines "queues", hence the name of this data structure.

The example of a waiting line appears all over. From modeling simple stores to factories, where items need to wait before being processed via a machine. From office clerks that need to process a stack of papers to call centers that need to assign callers to an employee. From scheduling problems to traffic jams. They appear everywhere and for good reason. Queues are very powerful and elegant structures for complex systems, which is also why we can clearly identify queues in [Ima87], [SIM94], [Fle93] and [INC97]. On top of that, the latter two also provide numerous different ways for stockpiling items in racks, warehouses, reservoirs (all of whom can be associated with a queue)...

First, let's take a look at some definitions and properties of queues and queueing systems. Afterwards, we will use this information to construct a valid mathematical model (i.e. a DEVS building block).

### 4.2.1 Queueing Theory

*Queueing theory* is the mathematical study of queues. We will define a *queueing system* as a set of multiple queues, combined with a queue selector.

#### Kendall's Notation

In [Ken53], Kendall defines a queueing system with three properties: the *input*, the *service-mechanism* and the *number of service channels* (i.e. the number of queues). The *input* defines how "customers"[11] arrive. More specifically, it defines which distribution the customers follow. The *service-mechanism* assumes the service time for successive customers is statistically independent from one another. In the same paper, Kendall also introduces a way to denote queueing systems:

$$A/S/c$$

where $A$ describes the arrival distribution, $S$ the service-mechanism used and $c$ the number of service channels. For both $A$ and $S$, the notations $D$ (deterministic or regular) and $M$ (random or Poissonian) are commonly used. For $S$, $G$ (generic or "nothing special") is also a common notation.

Since the publication of Kendall's paper in 1953, this notation has been extended to also include $K$ (capacity of the system, i.e. maximum number of

[11] Ships, airships, people, items...

customers allowed), $N$ (the calling source, i.e. the size of the population from where the customers come) and $D$ (the *queue-discipline*):

$$A/S/c/K/N/D$$

If omitted, these final three are respectively assumed to be $+\infty$, $+\infty$ and *FIFO*.

#### Queue-Discipline

Most waiting lines follow a very simple discipline: *first-come-first-served* (FCFS), which is better known as *first-in-first-out* (FIFO). Such a strategy describes the behaviour of the customers in a queue. For FIFO, the first customer that entered the queue will also be the first to leave.

Another famous discipline is the opposite, *last-in-first-out* (LIFO), which describes that the very last item to enter the queue will also be the first to leave. Within programming, this kind of queue is called a *stack*, because it acts like a stack of papers. You add a new paper to the top of the stack and when you're ready to process an item, you will start with the top-most paper first.

More generally, we can talk about a *priority queue*. Instead of defining a specific queue-discipline, a priority queue will assign a priority to each item that's enqueued. Dequeueing an item will dequeue the item with the highest priority first. The assigning of such a priority will be done via a *comparison function* $f_c$. Based on two inputs, the time $t$ ($t \in \mathbb{R}^+$) and a customer $c$[12], $f_c$ will determine the priority for $c$.

For instance, the priorities in a normal LIFO queue are the index (or timestamps) of the arrived items. For FIFO, we merely have to negate this value[13], i.e.

$$FIFO(t, c) \to -t \qquad\qquad LIFO(t, c) \to t$$

The generalization of the priority queue allows us to alter *Kendall's notation* for a general queueing system to the following:

$$A/S/c/K/N/f_c$$

#### Balking

Let's revisit the waiting line example. Nowadays most supermarkets have multiple cash registers where you can checkout your groceries. So there is a lot of choice in picking which waiting line to wait in. While there is an entire

[12] Since we want to be as inclusive as possible, $c$ does not have to be a number, but can be any object that we want to enqueue.

[13] Note that this is because we dequeue the highest priority first. If we were to dequeue the lowest priority first, these implementations must be swapped.

branch of psychology as to why you'd better pick one queue above another, most shoppers will just pick the shortest queue.

*Balking* is the making of a decision not to join a queue if it's too long. Furthermore, the selection of a queue in a multi-queue system can be identified via Liao's *balking index* [Lia11]. See also *figure 4.6*.

### Reneging

You're calling tech support because you have issues with your internet provider. Unfortunately, it's an issue on their side, causing a lot of people also to decide to call tech support. Every employee they have working there is currently taking a call, so you've been put on hold (i.e. you've been added to their queue).

Chances are, if you're impatient, after a while, you hang up before being processed. You basically leave the queue after a certain time delay, before being "processed". This principle is known as *reneging*.

Within our DEVS model, reneging will be an additional output from where items will exit the block. To decide which items need to be reneged, we can do one of three things:

1. Check the contents of the queue every once in a while. Even though this is the easiest solution, it is the least precise.
2. Assuming your model runs in seconds, you can check every second if there are items to renege. Alas, items that need to be reneged after, for instance, half a second will be reneged too late, presumable causing the system to experience unexpected behaviour.
3. The best solution (and also the one we will be using in our BBL) is to set our time advance $ta$ to the time $delay_{ren}$, when the next element should renege[14]. The reneging itself can then be implemented in $\delta_{int}$.

### Jockeying

As some sort of combination between balking and reneging, *jockeying* enters the stage. Imagine you're waiting in line for cash register A, but suddenly you realize the clerk at cash register B works faster. So you switch from queue A to queue B in the hope of being able to leave the store faster. It's this principle, switching a queue if the other one is shorter, that is called jockeying.

Basing ourselves on the jockeying implementation that ExtendSim introduces in [Dia+00], we will allow the last customer in our queue to jockey when it learns another queue in our system is preferred over the current one.

[14] If you allow elements to dequeue after a time delay $\Delta t$, you must set $ta$ to the minimum of $\Delta t$ and $t_r$.

**Faffing**
In supermarkets, you do more than just follow the queue. There are some waiting times to be had because of human interaction, payment and the *faffing* afterwards. Faffing is the name that was given to the time delay of gathering your things after paying at a checkout. According to [Kna10], this averages at about 3.17 seconds, but [Mel16] notes that Dan Meyer, chief academic officer at Desmos, puts this average at 41 seconds per person[15]. Either way, while we will not include this in our Atomic DEVS of a QUEUE, faffing must be taken into account for realistic simulations of supermarket models[16].

**Cutting Lines**
When waiting in line, it's pertinent the natural order (FIFO) is being held. When others disregard this order via *cutting the line*, this feels unjust and often allows arguments to be fueled. From your point of view, you are victimized by a *slip*, while we talk about *skipping* the line for the perpetrator (see [Lar87]). Within our model, we can use $f_c$ to implement this feature. For instance, instead of using pure FIFO, we might increase our priority at random in $f_c$, placing us at another location, closer to the front of the queue.

On the other hand, we can control which items may skip ahead. Think for instance about a cash register that prioritizes elderly people and pregnant women. Again, this may be implementing via manipulation of $f_c$.

**Queue Chunking**
Most amusement parks make their waiting lines interesting and visually pleasing. This is done because nobody likes to wait and it prevents jockeying and reneging. Often, *queue chuncking* is also added, where they seemingly break up the queue into smaller parts via the usage of walls. A person waiting in line will believe that their end goal is immediately behind this wall, which gives them hope and reduces the boredom of waiting. Once they reach the wall, they realize they were wrong, but see another possible end of the line.

Of course, in a purely mathematical model, this does not need to be added, but nevertheless it remains an interesting point of view for modelers that want realistic human psychology in queues. In fact, Walt Disney amusement parks go up and beyond to study waiting times and improve the user experience [Pri19]. Queue chuncking is one of the many tricks they employ to prevent reneging.

### 4.2.2 Building Blocks for Queues

We now have a good idea of what queueing systems look like and what they should do. Let's take a look at the building blocks we can create.

[15]In this case, it also includes saying hello and paying.
[16]Faffing can be obtained via using DELAY blocks.

### Queue

In *figure 4.4*, you can find a small Statechart for a simple queue (without reneging and jockeying). Each of the three states (`Enqueue`, `Dequeue` and `DequeueTimer`) can be seen as an orthogonal state, meaning they're executed in parallel.

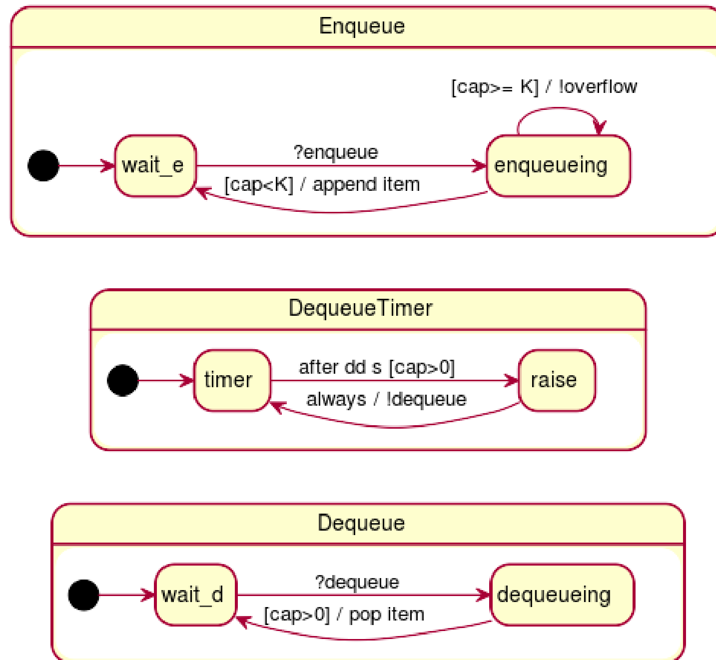
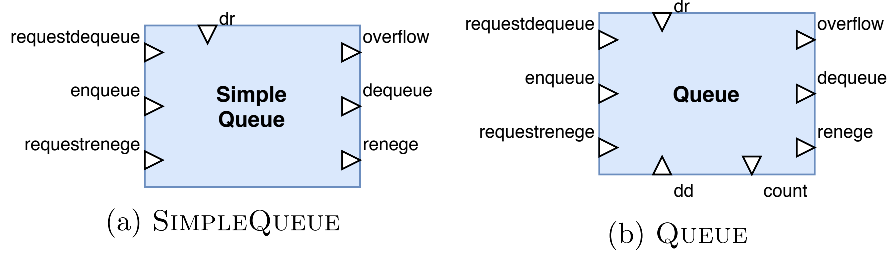



Figure 4.4: Simple Statechart for a QUEUE.

The first state, `Enqueue`, handles all enqueueing in our queue. As soon as we obtain an *enqueue* input event, the item will be appended if the current queue capacity *cap* is less than $K$. Otherwise, an *overflow* output event will be fired that passes on the item that needed to be enqueued. The `Dequeue` state is quite similar to the `Enqueue` state in that it will wait for a *requestdequeue* input (denoted as `?dequeue` in the figure) before dequeueing. This will only happen if $cap > 0$, ensuring that a *requestdequeue* on an empty queue will be remembered until the next item enters.

Finally, the `DequeueTimer` state will make sure the `?dequeue` event is triggered automatically after every *dd* time units. Obviously, this needs to be handled internally by the DEVS by linking the `!dequeue` to `?dequeue` within the block. This implies making use of a Coupled DEVS as shown in *figure 4.5*.

Now, let's delve deeper into the reneging part. Let's annotate each item $i$ that is enqueued with its absolute time of arrival $a_i$. From this value and $dr$, the reneging delay, we can easily determine the absolute time until the item needs to renege $a_i + dr$.




(a) SimpleQueue (b) Queue

Block 4.8: Simple Queue and Queue building blocks.

Let's say $T$ represents the current absolute time. The *ta* function can now be defined as the minimal value of all $dr - T + a_i$, if our queue is not empty, we're not overflowing and if no special item was requested[17]. The internal transition function, $\delta_{int}$, will handle the actual dequeueing and reneging, based on the current context.

Putting it all together, we will make two blocks: the Simple Queue (*block 4.8a*) and the Queue (*block 4.8b*). The former is a queue without automatic dequeueing (making use of *dd*), whereas the latter will implement *figure 4.5* as a Coupled DEVS.

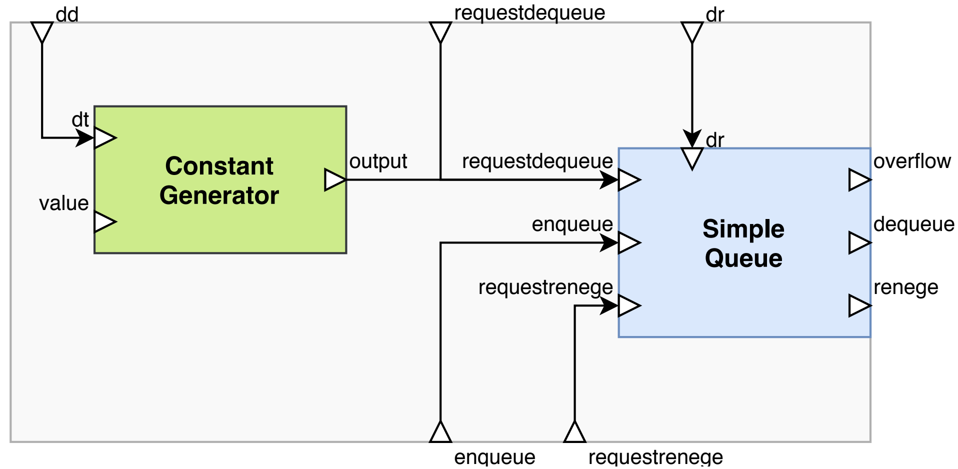



Figure 4.5: An overview of a Queue building block with automatic dequeues.

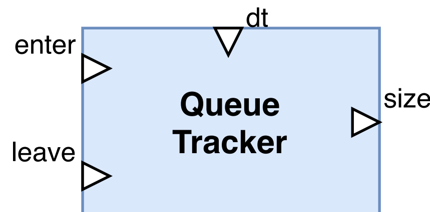
The *value* input for the Constant Generator will be ignored, because it does not matter which kind of message is inputted in the *requestdequeue* input of the Simple Queue.

The *requestrenege* input will request the queue to renege the last item in the queue (w.r.t. $f_c$). This mechanism is used in [Dia+00] to allow for jockeying.

[17] With a message on the *requestdequeue* or *requestrenege* input.

Now, if we look at the *Holding Tank* from [Ima87], we notice this block is able to output a variable amount of items. This can be simply implemented by changing the *requestdequeue* and *requestrenege* ports to integer ports, where the arriving message indicates the amount of items there are requested for output. Obviously, this is a seldom occurrence and this functionality must be mutable.

Finally, the QUEUE has one last issue that should be solved. Because of the way the CONSTANT GENERATOR works, the block will keep on running indefinitely. Because this does not make too much sense for a queue, let's use a brand new, specialized block: the QUEUE TRACKER (see *block 4.9*). It will replace the CONSTANT GENERATOR in *figure 4.5* and will be linked accordingly.




Block 4.9: QUEUE TRACKER specialized building block.

The QUEUE TRACKER basically tracks the current size of the SIMPLE QUEUE, which is outputted on the *size* output every time interval *dt*. An item that arrives on the *enter* input enqueues, whereas an item that arrives on the *leave* input leaves the queue via dequeueing, overflowing or reneging.

When the queue is empty, it will pause until the next item arrives. In Python(P)DEVS, this corresponds to setting the *ta* to $+\infty$.

Remember that we said that the SIMPLE QUEUE could have a number of items requested on its *requestdequeue* or *requestrenege* input ports? We have to keep that in mind when using the *size* output of the QUEUE TRACKER for a dequeue request. There are two options:

1. Either we say that no such functionality applies for a QUEUE,

2. or we say that the QUEUE TRACKER may also output a predefined value instead of the size.

The latter option provides the more generic functionality. When looking at the *Queue Tools* block from [Ima87] and the *Queue* from [SIM94], we can see that some models might require the queue to be partially filled on startup. Those contents could be provided via a constructor argument.

### Balking

Because the Queue block has a *count* output port, we can easily implement a basic balking example, as shown in *figure 4.6*. Whenever our Queue has more than 10 customers, all new customers leave the system on `finish 1` (see *section 4.7.1*, *block 4.31*). Otherwise, the customers will leave the system on `finish 2`. See also *section 4.1*, *block 4.1*; *section 4.7.5*, *block 4.33b* and *section 4.6*, *block 4.27*.

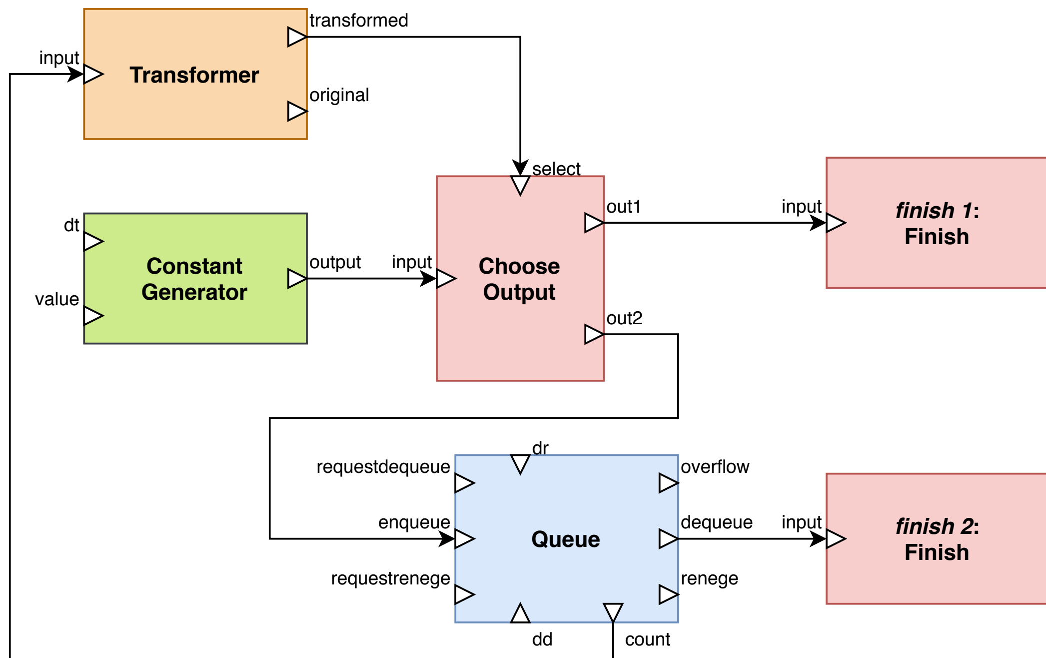
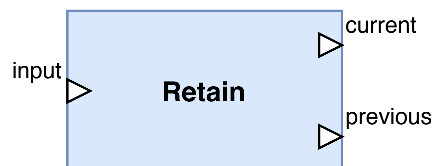



Figure 4.6: An example on balking.

### Remembering Items

Often, it can be useful for your model to remember certain items that passed through until the next cycle. Imagine you want to detect an infinite loop by counting how often the same item is being forwarded. This can be done using the Retain block (see *block 4.10*).




Block 4.10: Retain building block.

Its functionality is quite simple: every item that enters the *input* port, will be held until the next item arrives on this port. At the same time, when an

item arrives at this block, it will be outputted immediately on the *current* output and the previous item (if one exists) will be outputted on the *previous* output. Additionally, we will provide this block with two modes, that can be set upon block construction:

*individual:* Will act as described above for each item that arrives at this block.

*collection:* Will capture all messages that arrived within the same time unit, e.g. when $t_{elapsed} = 0$. The set of messages will be cleared when this is not the case. For each message that arrives, a list of messages will be outputted on the *previous* port.

**Advancing**

While already incredibly powerful, the QUEUE building block still lacks some functionalities. Let's take a look at an example to see what we're missing.

**Example 4.2.1** (Animal Rescue Service).

*In a nature reserve, the park rangers may find wounded animals. Whenever this happens, they take the animals to a local vet, so they can be nursed back to health. When the injuries were major, the animal will reside in an animal shelter, before being set free in the wild again.*

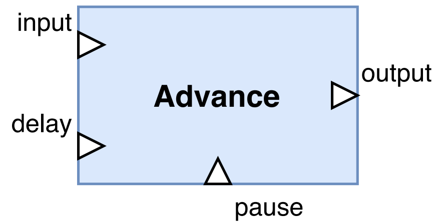
*Some animals are, unfortunately, more injured than others and therefore require a longer time to recover. We can state that both the inter-arrival rate of wounded animals as well as the recovery period follow a randomized distribution.*

Let's say we'll model this example using our current building block library. Obviously, we have a RANDOM DELAY GENERATOR for determining the arrival of animals. The shelter looks like a (SIMPLE) QUEUE that uses reneging, seeing as each animal must stay some time in the shelter, independent of the other animals. But how would we be modeling the randomness of their recovery period? The reneging time, $dr$, for the (SIMPLE) QUEUE is a static value that applies to all customers of the queue. Furthermore, changing this to another value during execution will also alter all other customers.

There are two solutions for this issue:

- Either, we change the (SIMPLE) QUEUE to make the $dr$ item-specific, or
- we add another building block that provides this functionality and acts independent of the QUEUE.

Seeing as the (SIMPLE) QUEUE is quite complex as-is, we'll opt for the second solution. Basing ourselves on GPSS for the name, the ADVANCE block (see *block 4.11*) is born.




Block 4.11: ADVANCE building block.

Whenever an item $i$ arrives in the ADVANCE block on the *input* port in the same timeframe as an input $\Delta$ arrives on the *delay* port, the item $i$ will be held for $\Delta$ time before being released on the *output* port. This allows items to wait for a random delay in a more flexible manner than the (SIMPLE) QUEUE did. Additionally, customers can skip ahead, past other items that are taking a longer time to finish. The *pause* input allows for a generic control of the customers in this queue[18] by providing a simple way of pausing the customers until further notice.

## 4.3 Gathering Data

More often than not, you want to be able to "capture" the items or data traveling through your model. This is useful for both statistical analysis and debugging purposes. In short: keeping track of data gives you a better insight into your models.

Data can be gathered using *collectors*. Collectors are small building components that gather information on arriving items. Either during simulation of your model or thereafter, the contents of the collectors can be analyzed in any way you see fit.

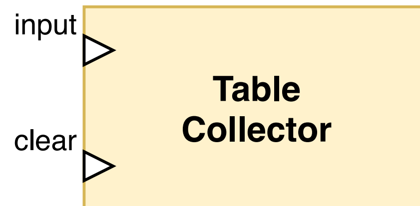
### 4.3.1 Tableization

A naive implementation would be the TABLE COLLECTOR (see *block 4.12*). This building block internally holds a table that is updated every time an item arrives on its *input* port. The table can have as many columns as are required within your problem domain, but let's focus on the basics. One of the columns is ideally the absolute time at which the item entered (and left) the collector. By itself, this can be used to identify bottlenecks in your system. Another column is obviously the item that arrives, as a whole. The item can be any

[18] Technically, it's not a queue anymore, but more of an *item pool*, seeing as no queueing discipline exists anymore, unless explicitly modeled by the input for the *delay*.

kind of data: numbers, characters, strings, lists, maps and even your own type of objects.

As you may be able to deduce, the TABLE COLLECTOR is a clean and powerful way of keeping track of all your data. But this is also the caveat. You are keeping track of *all* your data and your memory requirements will scale proportionally to the amount of items that have arrived.




Block 4.12: TABLE COLLECTOR building block.

[Ima87] has a *Clear Statistics* block, which clears all statistical information of the components. Because this might be useful, we also provide this feature by adding a *clear* input port.

### 4.3.2 Memory-Efficient Collection

The best way to circumvent the caveat of the TABLE COLLECTOR is to stop keeping track of *all* the data. Assuming your data is numerical and you desire to do a statistical analysis of the system, chances are you do not require much more than the *minimum* (*min*), the *maximum* (*max*), the *mean* ($\mu$), the *variance* ($Var$) and the *standard deviation* ($\sigma$). This corresponds to the *Min & Max* and *Mean & Variance* blocks from [Ima87].

The COLLECTOR[19] (*block 4.13*) does exactly that. It keeps track of these five values without the need for remembering all your data. How is this possible? We use basic statistical mathematics. We know the following formulas:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i \qquad Var = \sigma^2 = \sum_{i=1}^{N} \frac{(x_i - \mu)^2}{N}$$

Where $N$ is the amount of items that entered on the *input* port and $x_i$ the $i$th

[19] Called this way, seeing as this will be more commonly used over the other collectors.

Block 4.13: COLLECTOR building block.

item we see. The second formula allows for the following derivation:

$$\begin{aligned}
\sigma^2 &= \sum_{i=1}^{N} \frac{(x_i - \mu)^2}{N} \\
&= \frac{1}{N} \sum_{i=1}^{N} \left(x_i^2 - 2x_i\mu + \mu^2\right) \\
&= \frac{1}{N} \left( \sum_{i=1}^{N} x_i^2 - 2\mu \sum_{i=1}^{N} x_i + N\mu^2 \right) \\
&= \frac{1}{N} \sum_{i=1}^{N} x_i^2 - \frac{2\mu}{N} \sum_{i=1}^{N} x_i + \mu^2
\end{aligned}$$

Now, it is clear that, in order to keep track of the mean, the variance and the standard deviation, all we need is $N$, $\sum_{i=1}^{N} x_i$ and $\sum_{i=1}^{N} x_i^2$.

If we add two more values, *min* and *max*, we have reduced the memory-space from $N$ to 5, which can be updated every time there is an external transition $\delta_{ext}$ on the input *input*.

Additionally, we'll add a condition to the COLLECTOR's constructor, *positive*, which, when `true`, will store these values only if the number that's entering is strictly positive, i.e. if $x_i > 0$, an idea that's also used in [Cla92].

Similarly to the TABLE COLLECTOR, the gathered information can be reset upon the arrival of a message on the *clear* input.

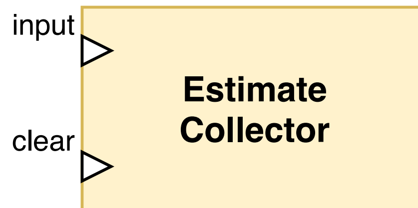
### 4.3.3 Colletion w.r.t. the $P^2$ Algorithm without Scoring Observations

In [JC85], an algorithm is proposed to heuristically calculate quantiles and histograms without the need to keep track of all data. Emphasis on "heuristic". Obviously it is impossible to perfectly predict the median of any sequence in a single formula that can be computed iteratively. Nevertheless, the proposal gets close to doing so.

The $P^2$ algorithm makes use of a set of $b+1$ markers, where $b$ represents the

amount of cells to look at[20]. When a new item $x_j$ enters the sequence, it tries to identify in which cell the new item would fall. Based on this information, the heights of the markers are updated via the piecewise-parabolic prediction ($P^2$) formula. In order to keep track of this information, we create an ESTIMATE COLLECTOR block that needs to call the $P^2$ algorithm every time their $\delta_{ext}$ gets called.

Assume $q$ is the list of the marker heights and $n$ the list of their positions, where $q_i$ indicates the $i$th height and $n_i$ the $i$th position. *Algorithm 2* gives the $P^2$ algorithm that the building block must execute every execution of $\delta_{ext}$. For the full algorithm, see [JC85].




Block 4.14: ESTIMATE COLLECTOR building block.

The ESTIMATE COLLECTOR block is shown graphically in *block 4.14*. The item that enters the block on its *input* port will call the $P^2$ algorithm before being implicitly discarded. When a message arrives on the *clear* input, the contents of the block will be reset.

### 4.3.4 Obtaining Numeric Data from other Items

As discussed previously, items do not have to be numeric in any way, shape or form. Yet, our COLLECTOR and ESTIMATE COLLECTOR are both building blocks that *only* work with numerical data. There are two ways of dealing with this issue:

1. You give each collector an additional unwrapping function $u(i, t)$ that takes two arguments: $i$, the item and $t$, the time that was passed at this point in time. Then, before applying all logic in the $\delta_{ext}$ functions, you simply "unwrap" the item beforehand.

2. Introduce an additional building block, the TRANSFORM block that also accepts such an unwrapping function $u$. It works as you might expect: an item enters this block and the unpacked result leaves it within the same time instance. See *section 4.6* for more information on this kind of block.

[20] $b + 1$ is the amount of "bars" in the histogram

**Algorithm 2** Simplified version of the $P^2$ algorithm step that needs to be called in every $\delta_{ext}$ of the ESTIMATE COLLECTOR block.

```
 1: procedure P2STEP(x_j, N)                    ▷ x_j is the new item, N items seen
 2:     if q doesn't have size b + 1 yet then
 3:         Add x_j to q and sort this list in increasing order
 4:     else
 5:         Find cell k such that q_k ≤ x_j < q_{k+1}
 6:         Adjust extreme values q_0 and q_{b+1} if necessary
 7:         n_i ← n_i + 1      ∀i ∈ {k, ..., b + 1}
 8:         for i = 0 → b do
 9:             d ← i · (N − 1) /b − n_i
10:             if (d ≥ 1 and n_{i+1} − n_i > 1) or
11:                 (d ≤ −1 and n_{i−1} − n_i < 1) then
12:                 d ← sign(d)
13:                 q'_i ← q_i from P^2 formula
14:                 if q_{i−1} < q'_i < q_{i+1} then
15:                     q_i ← q'_i
16:                 else                            ▷ Use the linear formula
17:                     q_i ← d · (q_{i+d} − q_i) / (n_{i+d} − n_i)
18:                 end if
19:                 n_i ← n_i + d
20:             end if
21:         end for
22:     end if
23: end procedure
```

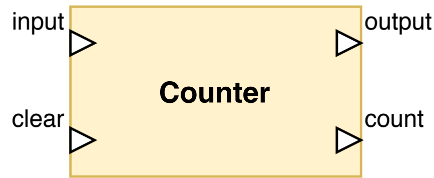
The latter choice not only provides a useful block within many contexts, it is also a cleaner and more streamlined way of obtaining such data, hence why this was chosen.

### 4.3.5 Counting Items

Let's imagine once more you're a store owner. You have made a model for predicting the peek hours and when you would best assign your staff where. Because you want to lure people to your store, you do a special deal: the thousandth customer will get a free shopping cart worth 100 euros.

Now, you want to know when this customer will be in your store, so you need to edit your model. We currently do not have a way of doing so *during* our simulation, which is why we will introduce the COUNTER building block, that counts the amount of messages that passed through. It is represented graphically in *block 4.15*.

Its functionality is quite simple: when a message arrives on the input *input*,




Block 4.15: COUNTER building block.

it will be outputted on the *output* port. Additionally, the amount of currently seen messages will be outputted on the *count* output. The *clear* input clears the statistics of this block, just as with any other collector.

## 4.4 Mathematics

Often it can be useful to be able to do computations during the simulation of your model. These can go from simple comparisons to full on $n$th degree polynomials. For single-argument functions (like most trigonometric formulas), the TRANSFORM block (see *section 4.6*) could be used, but we quickly hit the limitations of this block. To allow for all kinds of mathematical equations, we will create some building blocks, inspired by the Causal Block Diagram (CBD) [GDV16] formalism.

CBD is a formalism that allows the representation of mathematical formulas as a graph. This easily allows for a system of equations, which can be solved by the CBD simulator. While [VVV18] mentions CBD can be mapped onto DEVS, the solving of equations falls outside of the scope of this thesis, meaning we will not be bothered with so-called *algebraic loops*. Take a look at [Dem14] if you need to tackle this problem. Besides getting inspiration from the CBD formalism, we can also identify certain mathematical building blocks in [Ima87] and [LEG13].

### 4.4.1 General Functionality

We will be describing each individual building block later on, but before we do that, we must think about how these block will work. There are multiple possibilities:

1. All inputs must arrive at exactly the same timeframe. The output will be sent out immediately.

2. All inputs must arrive at the same time, but not necessarily within the same timeframe. For instance, a block outputs a value, which is sent

immediately to the input of the operator. Another block fires at the same time, but needs to be transformed before being inputted into the operator. The *ta* is 0 for all blocks in the chain, yet, both inputs do not arrive together.

3. The time of arrival does not matter. When all inputs have been arrived, the output is sent, otherwise, we keep waiting and remember all already-arrived inputs.

4. When there are inputs missing, a request is sent to obtain the next input.

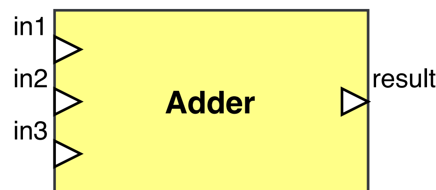
For the purposes of our library, we will consider option 3, but all options are valid, depending on what the end goal of the library is.

### 4.4.2 Basic Mathematics

In [Dia+00], there are numerous example models that make use of some very basic mathematical building blocks.

**Adder**

The ADDER (see *block 4.16*) has a $n$ inputs (*in1*, *in2*, ..., *inn*) and a single output *result*. As soon as all inputs are obtained, the sum is computed an outputted.




Block 4.16: ADDER building block.

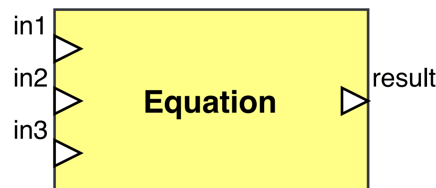
**Multiplier**

Similar to the ADDER, the MULTIPLIER (see *block 4.17*) has $n$ inputs (*in1*, *in2*, ..., *inn*) and a single output *result*. When all inputs are known, the product is outputted over *result*.

Block 4.17: MULTIPLIER building block.

### 4.4.3 Equations

We could go on and list every imaginable formula and/or equation, like CBD's *Subtractor*, *Divider*, *Inverter*, *Negator*..., but that would not be useful. All these blocks technically have the same functionality, which is why we will combine them all in a single block: the EQUATION block (see *block 4.18*).




Block 4.18: EQUATION building block.

The EQUATION block takes a finite set of user-defined inputs, applies an operation and outputs the result, respecting the conditions that were mentioned in *section 4.4.1*.

For instance, for a *Subtractor*, we define $x$ and $y$ as our inputs and $x - y$ as our operation. Whenever both $x$ and $y$ have been defined, the EQUATION block outputs their difference once, without delay. Additionally, we can provide a set of default arguments $D$ for our inputs, removing the necessity for all inputs to have obtained a value.

#### Solving Equations

Obviously, the EQUATION block only executes a function, given all inputs. It does not solve any equation with unknowns. When looking at [Ima87]'s *Data Filter* block, we realize this *does* allow for solving equations.

There exist numerous libraries to solve equations programmatically (most commonly by using some matrix techniques). For the purposes of this thesis, we will not divulge this matter any further.

#### Transformations

Within the provided context, it is not enforced that all inputs and the output of the EQUATION block are numerical. In fact, by not enforcing this, we'll allow more complex functionalities w.r.t. transforming data.

### 4.4.4 Complex Mathematics

Oft, it is also useful to be able to do more complex mathematics. Besides the possibility to chain the already described building blocks together (and thus obtaining an even bigger equation), we might want to have some additional functionalities within these mathematical blocks.

**Differentiator**

CBD allows you to use differentials (over the time) within your models. This is a feature that can be used to determine the rate change throughout time. It closely corresponds to the differentiator that can be used in electronic circuits.

To allow for a similar feature, we will introduce the DIFFERENTIATOR block (see *block 4.19*) that yields an approximation of the differential of the rate change. We will make use of the differential definition for the backwards difference to describe the output of this block:

$$\frac{d}{dt}f(t) = \lim_{\Delta t \to 0} \frac{f(t) - f(t - \Delta t)}{\Delta t}$$

where $t$ is the current time and $\Delta t$ the elapsed time between the previous input and $t$. Ideally, we would want $\Delta t$ to be as small as possible, but we leave this up to the user. The closer to 0, the more precise the result.

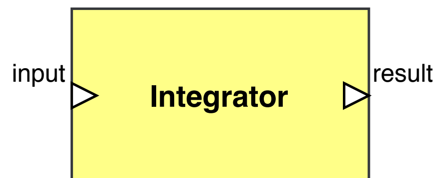
Block 4.19: DIFFERENTIATOR building block.

Therefore, each item $x_i$ that arrives at the *input* port will use the previous value (or the *initial condition IC* if no such value exists) $x_{i-1}$ and the elapsed time $\Delta t$ to output $v$ on the *result* port:

$$v = \frac{x_i - x_{i-1}}{\Delta t}$$

**Integrator**

If we can differentiate, we should be able to integrate as well. Hence, we introduce the INTEGRATOR block (see *block 4.20*). Strangely enough, [Ima87] has an *Integrator* block, but lacks a *Differentiator*.




Block 4.20: INTEGRATOR building block.

In order to determine which value to output, we will be using the assumption that our input comes from a continuous function and our delay between inputs $\Delta t$ is as small as possible.

Block 4.21: RANDOM building block.

To compute the integral at time $t$, when $\Delta t$ time has elapsed, we can use:

$$v(t) = \int f(\tau)\ d\tau = IC + \int_0^t f(\tau)\ d\tau = v(t - \Delta t) + \int_{t-\Delta t}^t f(\tau)\ d\tau$$

where $v(t - \Delta t)$ was the previous value of the integral. Additionally, we want $v(0) = IC$, or, in other words, the very first value that arrives in the block should output $IC$. Hence, we know $f(t)$, $f(t-\Delta t)$ and $v(t-\Delta t)$ at every time $t$ ($t > 0$). There are three methods to estimate the remaining unknown in the equation:

$$\int_{t-\Delta t}^t f(\tau)\ d\tau = \begin{cases} \Delta t \cdot f(t - \Delta t) & \text{lower bound} \\ \Delta t \cdot f(t) & \text{upper bound} \\ \frac{\Delta t}{2} \cdot (f(t - \Delta t) + f(t)) & \text{trapezoidal rule} \end{cases}$$

Alternatively, Simpson's rule could also be used if we keep track of three points and if $\Delta t$ remains consistent for all consequent calls of the block:

$$v(t) = v(t - 2\Delta t) + \frac{\Delta t}{3} \cdot (f(t - 2\Delta t) + 4f(t - \Delta t) + f(t))$$

Because we have no guarantee that $\Delta t$ will remain consistent, Simpson's rule is not implemented in PythonDEVS-BBL. The other rules can be defined by the user. Whenever a value arrives on the *input* port, that value is said to be $f(t)$ and $v(t)$ is computed before it's outputted on the *result* port.

**Randomized Values**

In *section 4.1.2*, the generation of random numbers was discussed, resulting in the RANDOM NUMBER GENERATOR. While this works for continuously generating such building blocks, we often require to associate a random number to each passing item, instead of to each time instance. Think of the *Random* block from [LEG13] and randomized arrival times or delays in [Ima87], [SIM94], [Fle93] and [INC97].

Using the same logic as in *block 4.4*, we can create a RANDOM block (see *block 4.21*) that outputs a random value on its *random* port as soon as it obtains an input. The inputted value will also be outputted on the *value* port. The distributions mentioned in *appendix A.2* can also be used here.

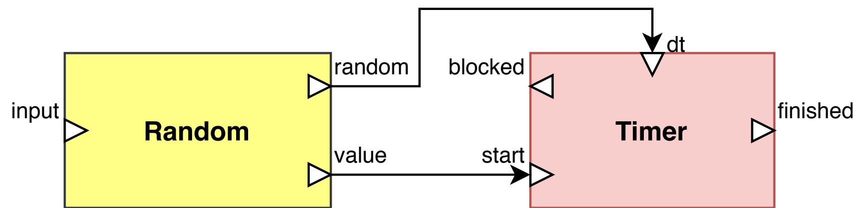



Figure 4.7: Creating a randomized delay from the RANDOM and the TIMER blocks.

*Figure 4.7* shows how the RANDOM block can be used in combination with the TIMER (see *section 4.7.8*, *block 4.37*) to create a randomized delay for each message, just like [Ima87], [SIM94], [Fle93] and [INC97] can do.

The DEVS model in the figure also corresponds to the RANDOM DELAY GENERATOR in the same way that the RANDOM block corresponds to the RANDOM NUMBER GENERATOR. That being said, it is probably preferred to use the ADVANCE block (see *section 4.2.2*, *block 4.11*) instead.

## 4.5 Input/Output

Working with a computer system often requires user interaction. Either by outputting debug information to some file or console, or maybe by asking the user for input. This not only helps in making a system that will work hand-in-hand with its users, but also for tracing issues and problems.

The DEVS formalism does not have a predetermined way to handle I/O information. An output simply corresponds with a message that is outputted from a block. An input, on the other hand, is a message that enters a block and presumably changes the block's state.

Note that Python(P)DEVS also provides a way to listen to and interrupt a real-time simulation. These should be used for actual user interaction in real-time systems, whereas the building blocks that are to be discussed in this section define a way to get a more generic interaction that resonates with a programmer's mindset.

### 4.5.1 Output

A standard for message logging, called syslog is given in [Ger+09]. It allows for a general way to store, parse, read and create messages throughout software systems, in a platform-independent manner. Within syslog, all messages are given a *facility* and a *severity* value, which are normative, but often used.

| Numerical Code | Facility |
|---|---|
| 0 | Kernel Messages |
| 1 | User-Level Messages |
| 2 | Mail System |
| 3 | System Daemons |
| 4 | Security/Authorization Messages |
| 5 | Messages generated internally by syslogd. |
| 6 | Line Printer Subsystem |
| 7 | Network News Subsystem |
| 8 | UUCP Subsystem |
| 9 | Clock Daemon |
| 10 | Security/Authorization Messages |
| 11 | FTP Daemon |
| 12 | NTP Subsystem |
| 13 | Log Audit |
| 14 | Log Alert |
| 15 | Clock/Scheduling Daemon |
| 16-23 | Local Use 0 - 7 |

Table 4.2: The facility code table for syslog messages.

*Tables 4.2* and *4.3* give an overview of these values[21].

Depending on the severity code, your system might require to exit execution. This can be used to throw errors that break a setup or cause an inconsistency. To indicate when your system requires this exit, a threshold is set. If you have a system where the occurrence of such a message is really dangerous (like for cars, airplanes, banking applications, factories...), this threshold will be high. If this is not too big of a deal (as, for instance, in supermarkets, resource allocation...), this value is usually low. But, there is no general rule. The higher this threshold, the less acceptable you are for the occurrence of errors.

To allow for syslog in our library, we introduce the following nine blocks: Logger, Emergency, Alert, Critical, Error, Warn, Notice, Info and Debug. All their representations are shown in *block 4.22*.

The Logger is a block which is able to log to any level of the user's choosing. This allows for a fast interchangeability in the model, but requires the knowledge of *table 4.3*. All other blocks "inherit" from this block and

[21]Note that in Python(P)DEVS, the default facility is 19.

| Numerical Code | Severity | Description |
|---|---|---|
| 0 | Emergency | System is unusable |
| 1 | Alert | Action must be taken immediately |
| 2 | Critical | Critical conditions |
| 3 | Error | Error conditions |
| 4 | Warning | Warning conditions |
| 5 | Notice | Normal, but significant condition |
| 6 | Informational | Informational messages |
| 7 | Debug | Debug-level messages |

Table 4.3: The severity code table for syslog messages.

statically assign their corresponding level to all messages they must sent. All loggers have a single input port that accepts the message to log. Internally, the block transforms the string to a valid syslog message and logs this to the available syslog server. The facility code that must be used is dependent on the context of the models.

Note that, by default, Python's `logging` module does not implement RFC 5424 [Ger+09] and therefore Python(P)DEVS doesn't either. For instance, logging to levels 0 (*Emergency*), 1 (*Alert*) and 5 (*Notice*) are not enabled by default because [Pyt] claims they are not too often used. The `pypdevsbbl.generic.io` module from the implementation of the BBL provides a `setLogger` function that, by default, follows the RFC. Users may opt out of this functionality by setting the `rfc5424` argument to a falsy value.

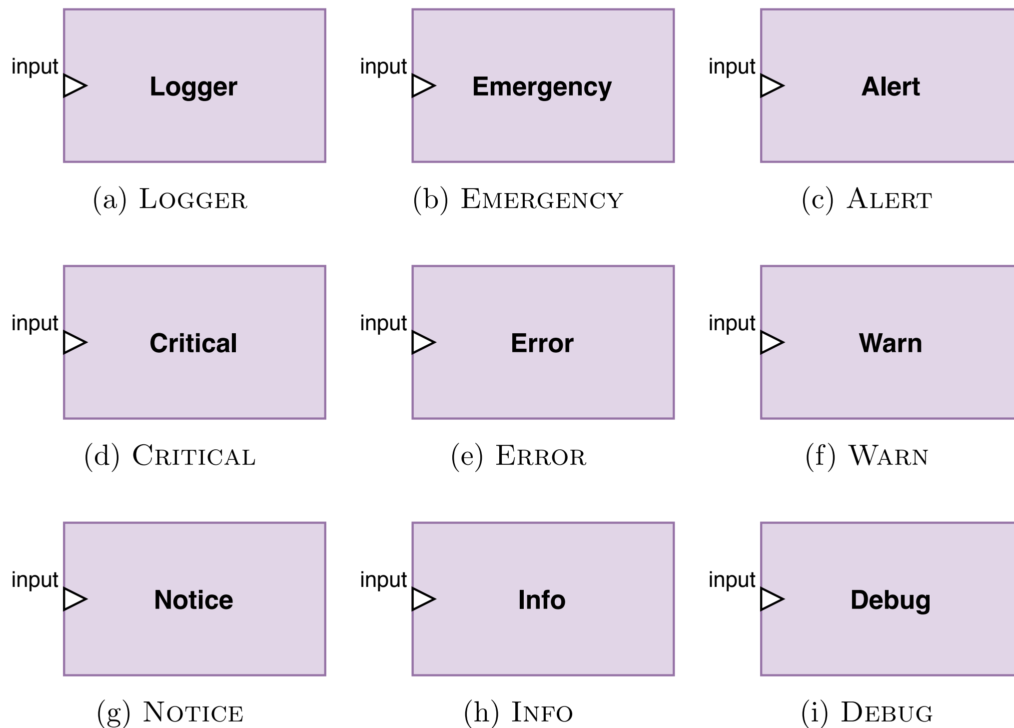
For those that have difficulties setting up a syslog server to listen for the logged messages, a simple to use and lightweight syslog server for Linux, created by the author of this thesis, is available on https://github.com/RandyParedis/PySysLogQt.

### Writing to Files

Another way of generating output is by writing free text to files. The syslog solution described before will output messages in a predefined format. While this is extremely useful for application-independent situations, it might not always be preferred. Often, you just might want files to be generated according to your own format or domain-specific language (DSL).

The File Writer block (as shown in *block 4.23*) will allow you to do so. It has a single input port on which it accepts a string that will be written to a file as-is. If we look at the Python documentation [Pyt], we can deduce there are numerous ways to open a file[22]. For the purposes of our library, we will

[22] Just like all major programming languages.

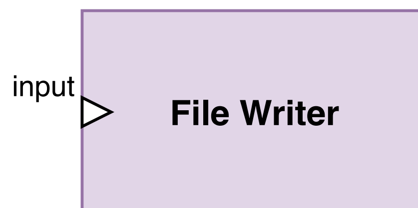



(a) Logger (b) Emergency (c) Alert

(d) Critical (e) Error (f) Warn

(g) Notice (h) Info (i) Debug

Block 4.22: Representation of the Logger building block and its derivatives.




Block 4.23: File Writer building block.

only focus on three of them. The first two (*write* and *binary*) can only be used exclusively, i.e. they cannot be used together and at least one must be set. In the implementation of the File Writer a boolean value is therefore used. The last flag (*append*) can be used in combination with the previous two.

*write* Open for writing textual data, truncating existing data.

*binary* Open in binary mode, i.e. do not read the contents as a string, but more as binary data. Truncates the existing data.

*append* Append new input to the file instead of overwriting it.

Because the usage of files often leads to memory leaks, it is preferred to open and close the file if (and only if) a message needs to be written.

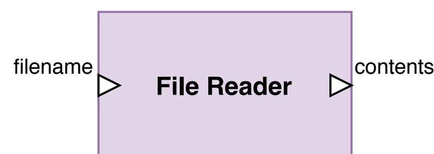
### 4.5.2 Input

[Ima87] makes use of a *Popups* block that temporarily interrupts the simulation while it waits for the user to enter input. Within DEVS, this kind of user interaction cannot and should not be done. Even if it were possible to describe this within the formalism, it counteracts what we're trying to achieve with real-time simulations. Asking for user input will always pause or halt our system and therefore we loose our wall-clock time. Hence, no corresponding block exists within our library. If this functionality is required, you can make use of a LISTENER (see *section 4.5.4*) that tracks all user input.

**Reading from Files**

Reading from files is more difficult than writing. If you have predefined your own file format, you must parse it again before you can use it. There is also the possibility to write a binary file, so let's add a way to read one as well.

The result is a FILE READER that has a string input *filename* and an output *contents* (see also *block 4.24*). Every time there is a new file to be read, its filename must be send on the block's input and the contents will be outputted. With a constructor flag, you can identify if you're reading binary data or not.



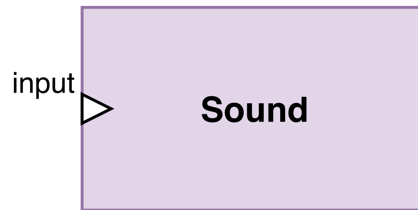

Block 4.24: FILE READER building block.

You can use the TRANSFORM block (see *section 4.6*) to fully parse the read data into a format you desire. CSV files are quite simple to parse, whereas complex file structures may require the creation of a grammar, so it can build an *abstract syntax tree* (AST) from your file contents. It is believed this falls outside of the scope of this project, but if you require a lot of file parsing, it might be a good idea to create a custom PARSE block. You may be able to subclass the TRANSFORM block as a starting point.

### 4.5.3 Playing Sounds

Both [Ima87] as [LEG13] provide a way to play sound as some sort of feedback, respectively in the *Notify* and the *Sound* blocks. As such, the SOUND block (see *block 4.25*) was born. Whenever it receives an input, it will conditionally play a sound. The sound file and the condition are both given as constructor arguments.

Note that some additional libraries might be necessary for this block to work. While this can be enforced, PythonDEVS-BBL doesn't. Instead, a warn-

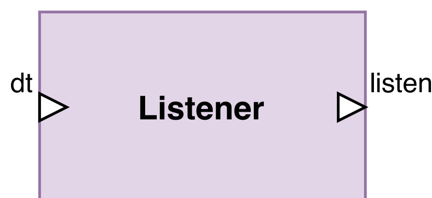



Block 4.25: SOUND building block.

ing will be shown if the library could not be located. Within the implementation provided for this thesis, `pydub` [Rob11] and `simpleaudio` [Ham15] are required to be installed if the SOUND block needs to be used. If it doesn't, the mentioned libraries do not need to be installed. Because of `pydub`, any file format supported by `FFmpeg` [Bel11] can be used in this block and it is perfectly possible to indicate a segment of the sound to play (in milliseconds). This can be useful if you don't want a massive library of sound files, but rather snippets out of one single file. This can also be used if it appears your sound takes way too long to complete. Additionally, `simpleaudio` will allow the sounds to be played asynchronously, meaning they won't halt the execution of the simulation until they have finished playing.

### 4.5.4 Listening to External Events

Often, it might also be useful to listen to events that happen externally. Think about the sensors or the *Bluetooth Connection* in [LEG13]; or the *Button*, *Link Alert* and *Switch* blocks from [Ima87].




Block 4.26: LISTENER building block.

This can be solved by introducing a LISTENER building block that continuously polls for changes to an external event. Every $\Delta t$ time, the LISTENER calls a polling function $f_p$ that takes at least one argument: the simulation time. Its return value is outputted on the *listen* port.

The power of this block comes from the fact that $f_p$ can either read from a file to get an updated value, read the current sensor value in a blocking way, or capture data from an external process.

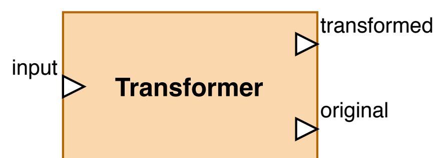
## 4.6 Transforming Data

The items that are traveling though your system do not have to be the same type. It is perfectly possible one building block is sending a string whilst another is sending a number. But why stop there? One of the core concepts of object-oriented programming is the possibility for having multiple *objects*. Such an object is a structured piece of memory that can be used or called upon elsewhere.

It is perfectly understandable that you want to have your own type of items traveling through the system, i.e. a `Crate` that is being transported through your model of a factory, or a `Car` that is slowly being built on the conveyor belts. If you could only use numbers for this kind of systems, you would run into issues really fast, which is why it is important for items to be able to be transformed. We distinguish three kinds of transformers: the TRANSFORMER block, the PACK block and the UNPACK block, all of whom have particular purposes.

### 4.6.1 Transforming via Functions

It is ever so useful to be able to transform one data type into another. This is, for instance, useful when turning complex objects into numerical data to compute mathematical problems, as can be required by some blocks discussed in *sections 4.3 and 4.4*. Generally, it is possible to state that, given a transformation function $T(i, e, t)$, where $i$ is the item, $e$ the elapsed time since the previous event and $t$ the simulation time, you can define a way to turn any item into any other using an algorithm.




Block 4.27: TRANSFORMER building block.

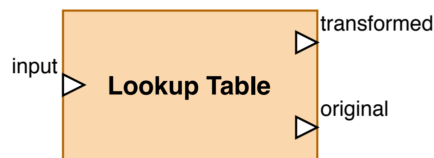
We will introduce a TRANSFORMER block (see *block 4.27*) that takes $T$ as a constructor argument. Every item it receives on its input port (*input*) will be automatically transformed through $T$ and outputted immediately on the output port *transformed*. The item itself will be outputted on the *original* port, which allows this block to be located in the middle of a block chain.

Note that, the more complex the algorithm, the longer it will take to execute this block. This is especially important to keep in mind when we concern

ourselves with real-time simulations and wall clock time. When multiple inputs are required, the EQUATION block (see *section 4.4*, *block 4.18*) could be used instead.

### 4.6.2 Lookup Table

Because it's perfectly possible that not all input values can be mapped onto another value using a function, we will follow [Ima87]'s example and introduce a LOOKUP TABLE block. This block can simply inherit from the TRANSFORMER block, where the function is a mapping from the provided table. Therefore, it should come to no surprise that it looks and acts exactly the same, as you can see in *block 4.28*.




Block 4.28: LOOKUP TABLE building block.

### 4.6.3 Packing and Unpacking

Another way of transforming data is to group multiple items together and split them up again afterwards. These are processes that are known under many names. [Ima87] uses "*batching*" and "*unbatching*", while [Fle93] describes it as "*combining*" and "*separating*". [INC97] makes do with "*assembling*" and "*unpacking*", but we will refer to it as "*packing*" and "*unpacking*".

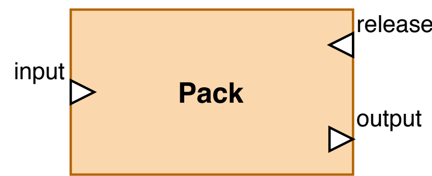
#### Making Packages

The PACK block is a simple block (see *block 4.29*). Based on a predefined `Package` class[23] that defines what happens to new items in the package, all items that enter the block on the *input* port, will call a transformation function as is defined in the `Package` class. This class is set upon construction of the block and corresponds to the *Container* atom from [INC97].

The `Package` class has two main methods: `pack` and `unpack`. The former takes three arguments: *new* (the new item to add), *elapsed* (the elapsed time) and *time* (the simulation time). Moreover the `unpack` method later.

Finally, if a *release* was obtained, the PACK block releases the held item over the *output* port and reverts to the initial state (where an empty `Package` is being held). Alternatively, because we use a specified class, we can also make use of a `finished` method in that class. Whenever this method evaluates to `true` upon acquisition of an input, it will be released automatically. In

[23] `pypdevsbbl.extra.packaging.Package`

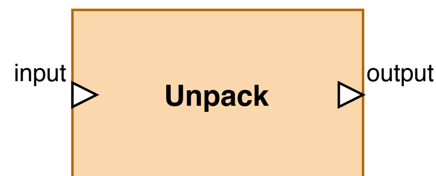



Block 4.29: PACK building block.

practice, a specific subclass of the `Package` class can be used to allow it to be applicable to the `Car` example given earlier.

**Extracting Packages**

The counterpart of the PACK block is the UNPACK block, which assumes all items that arrive on its *input* are of type `Package` and therefore have an `unpack` function that yields a list of all items that can be unpacked. This list is outputted on the *output* port. A graphical representation of this block is given in *block 4.30*.




Block 4.30: UNPACK building block.

## 4.7 Routing

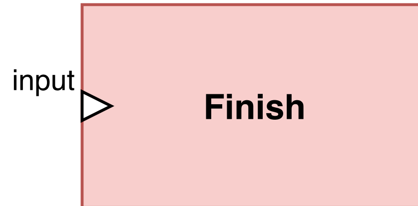
Often, messages do not follow a straightforward path throughout your model. There can be branches, splits, joins... This section describes a way to route your messages through your model.

### 4.7.1 Exiting

Items that are traveling through our system might want to exit it. This happens when, for instance, clients leave a supermarket, or assembled items are finished. All tools from *section 3.1* have this kind of feature. The *Exit* block is used by [Ima87] and the *End Point* block appears in [SIM94]. [Fle93] and [INC97] have a *Sink* and even [LEG13] provides a *Stop Program* block[24]. For

[24] As mentioned in *section 3.1.5*, [LEG13] is actually a Flowchart, meaning that this block also indicates the end of the program.

the most part, these blocks make sure the items are destroyed from the simulation and additional statistics can be gathered. As far as the statistics are concerned, the reader is referred to *section 4.8* for more info. Thus, we'll only look at the destruction of messages.

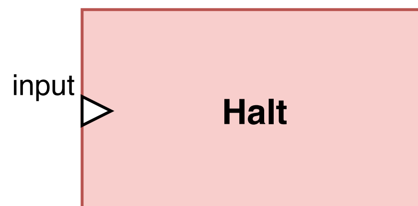



Block 4.31: FINISH building block.

The DEVS formalism does not require such a feature. All messages exiting an output that is not connected to anything will be silently discarded. Nevertheless, the addition of a FINISH block (see *block 4.31*) provides a clear endpoint in a model, both for the modeler as for anyone that needs to read or decipher the model. The FINISH block destroys an arriving message from the simulation[25]. Whenever you need to gather additional statistics, it is preferred to use collectors (see *section 4.3*) instead.

### 4.7.2 Terminating a Simulation

Sometimes, it might be useful for your simulation to terminate when a certain condition is met. Often, this condition can be set in the simulator itself, but you might also require to end a simulation if a message enters a block, similar to the *Stop Program* block from [LEG13]. For instance, you might want to terminate your simulation as soon as a queue starts overflowing.




Block 4.32: HALT building block.

While such a situation is merely an event that's fired, we will introduce a HALT block (see *block 4.32*), which provides the above-mentioned termination condition. Additionally, it will store the message that arrived before halting, so an end-user may use this for analysis.

[25]Technically, in PythonDEVS-BBL this block does nothing, seeing as Python has automatic garbage collection.

In PythonDEVS-BBL, the static function `anyHalted` can be used as a termination function on the simulation `sim`:

```
sim.setTerminationCondition(Halt.anyHalted)
```

Do note that the `anyHalted` function is slow if the amount of building blocks in the model is high[26]. Whenever all HALT blocks in the model are known and stored in the `halt_blocks` list, the following (faster) code can be used instead:

```
sim.setTerminationCondition(lambda m, t:
        any((h.isHalted() for h in halt_blocks)))
```

### 4.7.3 Exceptions

[Ima87] provides a *Throw Item*, a *Throw Value*, a *Catch Item* and a *Catch Value* block to jump from one location to another. While this is useful when you have a lot of connections, it does not follow the CDEVS semantics[27].

Additionally, an *exception* in DEVS is merely an event that is fired. To print exceptions, the loggers (*section 4.5.1*) can be used. This can be combined with the HALT block (*section 4.7.2*) if the simulation needs to be terminated whenever a condition is reached.

### 4.7.4 Splitting and Joining

In DEVS, it does not matter that multiple messages arrive at the same port, at the same time, especially in PDEVS. Similarly, when an output connects to multiple inputs, the message is duplicated automatically. While this makes it so we do not have to bother with specific splitters and joiners, it is good to keep in mind that the message was duplicated.

#### Block Outputs

This might seem quite contradictory when some building blocks will output the entered message as well. For instance, take a look at the COUNTER (*block 4.15*) and the TRANSFORMER (*block 4.27*).

In fact, there is a reasoning for those outputs. As mentioned in *section 2.2.7*, CDEVS are evaluated at each output event, not at each time instance. To prevent some of the issues that are yielded from this construct, the output ports under discussion make sure that the block outputs happen in the same timeframe, which can be used for multiple computations at the same time. More information can be found in *section 4.7.9*.

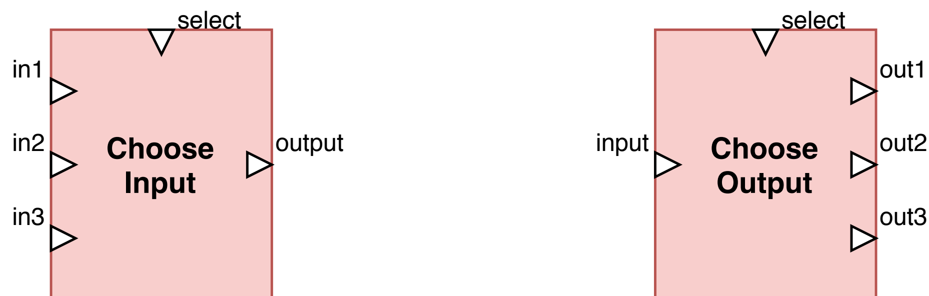
[26] As a sidenote, the Python(P)DEVS documentation states that the usage of a termination condition function slows down the simulation.

[27] Nor the PDEVS semantics for that matter.

### 4.7.5 Conditionals

Despite of what *section 4.7.4* might suggest, there is still some way to handle multiple paths converging or splitting. For this we return briefly to a telephone switchboard that was used from the late 19th century up to the second half of the 20th century. Back in the day, when you had to call someone, you had to tell the telephone operator who that was. They would link the connection through to a switchboard in such a way that it corresponded to your call. This was done by plugging a cable that made a connection into the board. Later on this was automated into the dialing system we still know.

#### Building Blocks




(a) Choose Input building block. (b) Choose Output building block.

Block 4.33: Choose Input and Choose Output building blocks.

Let's create two building blocks that would allow for such a system: the Choose Input and Choose Output blocks (see *blocks 4.33a* and *4.33b*). Based upon the value that arrives on their *select* input port, a specific path is opened. The message that arrives on an input other than *select* will be sent out of the corresponding path. All other messages will be discarded.

Note that, if the Choose Input or Choose Output blocks have two paths, mapped on 0 and 1, the *select* input can be a truthy value, which allows for boolean checks. There is only a single inconvenience remaining within these blocks: what happens when the value from *select* does not correspond to a valid path? The *Select Value In*, *Select Value Out*, *Select Item In* and *Select Item Out* blocks from [Ima87] indicate that there are some possibilities. We opt for a *round robin* system, i.e. we do a modulo division on the selected value to get a value that falls within the range of the possible paths.

#### Choose Based on Message

For the Choose Output block, we might not always want to choose our path based on the value of the *select* input. Often, the value that must travel the path might want to indicate which path to follow. Think for instance about the *Decision* block from [Ima87] or the *Switch* block from [LEG13].

With the given block this may be achieved by first extracting the value that indicates the output and send it to the *select* input at the same time that the input enters the block. An example on how to do this is given in *figure 4.8*. To make sure this interaction happens correctly, the $\delta_{ext}$ must first evaluate the new *select* input (if one exists) before sending the message out via its path.

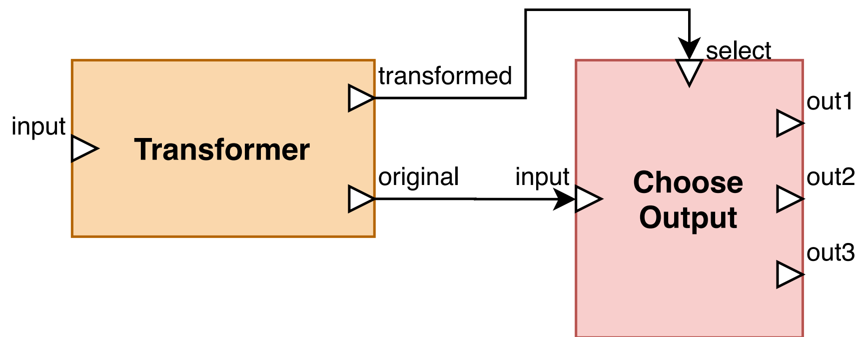
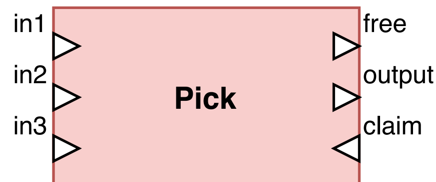



Figure 4.8: Example of how to set the *select* input of the CHOOSE OUTPUT block by default.

### Busy and Free Paths

Let's say that the outputs of a CHOOSE OUTPUT block are linked to a resource that can be *busy* or *free*. Whenever it's busy, no other item may obtain access to the resource. Currently, we have no way of doing this[28]. So let's introduce the PICK block, as shown graphically in *block 4.34*.




Block 4.34: PICK building block.

It is aware of multiple resources that can be free or busy and coordinates the information with the SELECT OUTPUT block that handles access to the resources. In the block, there are three ports that are always present: *free*, *output* and *claim*. The PICK block outputs `true` over the *free* port whenever it knows that there is at least a single resource that's free. This port can be used to signal a (SIMPLE) QUEUE (see *section 4.2*, *blocks 4.8a* and *4.8b*) that the resources are waiting for their next item(s).

Not to be confused with the *free* port, the *output* port outputs the identifier of a path to the *select* input of a CHOOSE OUTPUT block. Whereas *free* and

[28]Not in a clean fashion, that is.

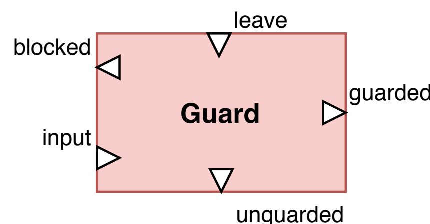
*output* were output ports, the *claim* port is an input that notifies the PICK block a resource has become unavailable.

Finally, the block also has $n$ input ports (equal to the amount of outputs from the CHOOSE OUTPUT block). When there is a message that arrives here, its corresponding resource will be marked “free”.

In the constructor, we’ll also add the possibility to mark certain resources as “busy” on startup and we provide a way to select the next resource, based on the list of all resources and their idle times. For completeness’ sake, we’ll also give the user the possibility to indicate that the last possible path is to be the fallback path (i.e. the path that’s selected when all resources are busy).

### 4.7.6 Guards

Often, users might want to shield a section of the model from other input. This is often the case when talking about *critical sections* on multi-threaded machines. [Ima87] and [INC97] provide multiple blocks that allow for acquiry and release of resources. In order to allow our DEVS library the same functionality, we could follow [INC97] and create a *Lock* and *Unlock* block, but in DEVS, they would require to be connected to one another. This can be done more easily in a single GUARD building block (see *block 4.35*) with two inputs (*input* and *leave*) and three outputs (*blocked*, *guarded* and *unguarded*).




Block 4.35: GUARD building block.

The GUARD block has an internal counter $n$ that tracks the amount of items that are being guarded. It starts at a predetermined value and, if it’s larger than 0, it decreases every time an item arrives on the *input* port. Afterwards, the item is outputted via the *guarded* output. If $n$ is 0, the item that arrived is “bounced back” via the *blocked* port.

When a message arrives on the *leave* input, $n$ is increased and the message is outputted on the *unguarded* output. Additionally, the block has two functions: $w$ and $u$ that both take an item and should return the weight that that item requires. Even though the GUARD block may be implemented as a Coupled DEVS, the functionality was specific enough to decide it’d be best to place it inside of an Atomic DEVS.

### Following a Queue

Even though items may be bounced back, often it is better for them to be waiting in a QUEUE, before they enter the critical section. In order to model this correctly in our DEVS library, we have to make use of the GUARD, QUEUE (see *section 4.2*) and SINGLE FIRE (see *section 4.1.4*) blocks, as illustrated in *figure 4.9*.

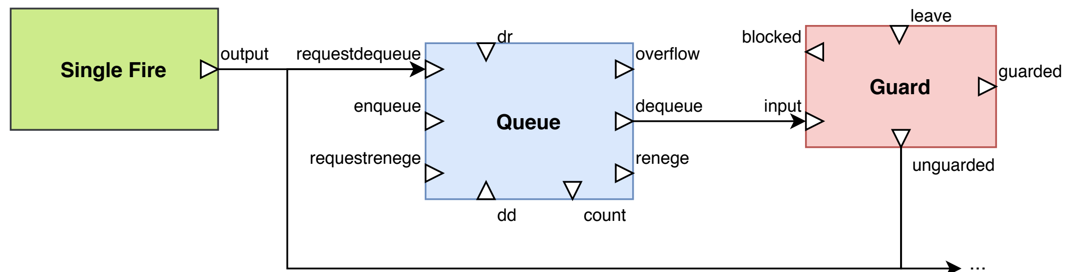



Figure 4.9: Using the GUARD, QUEUE and SINGLE FIRE blocks to create a queue before a critical section.

The QUEUE is constructed so it will not automatically dequeue items. At time 0, the SINGLE FIRE block will send a message on the *requestdequeue* of the QUEUE, meaning that it will immediately pass on the first enqueued item. Whenever an item leaves the guarded section, another dequeue event is requested.

You should make use of the TABLE GENERATOR (see *section 4.1.1*) with multiple records at time 0 instead of the SINGLE FIRE block if the $n$ value of the GUARD is larger than 1. Additionally, the SIMPLE QUEUE (see *section 4.2*) can be used instead of the QUEUE if you don't require the QUEUE's *dd* input.

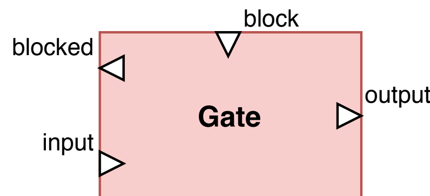
### Ambiguity

There is still a problem: what will happen first in the GUARD? Blocking or leaving of items? I.e. what happens if an item leaves at exactly the same time that another requests access? It makes the most sense that the leaving action will be processed before the optional blocking action. This would free a set of resources before new access is requested.

Whereas this might seem enough to remove the ambiguity in the block, it unfortunately isn't. Because we're using CDEVS, we cannot be certain that an input on the *leave* port and one on the *input* port happen in the same timeframe (i.e. the same execution of $\delta_{ext}$, see also *section 2.2.7*). This issue can be solved by using a SYNC block (see *section 4.7.9*).

### 4.7.7 Gates

Whereas using the Guard block can become complex quite fast, the Gate block (see *block 4.36*) tries to simplify some of it. If a part of your model requires toggleable access, it is recommended to use the Gate over of the Guard.




Block 4.36: Gate building block.

Instead of using resources to allow access to a critical section, the Gate accepts a boolean message on the *block* input. When `false`[29], all following messages that arrive on the *input* will be outputted on the *out* port, until *block* receives `true`[30], meaning that all messages should instead be outputted on the *blocked* port.

#### Shifts

One main use of the Gate block is its possibility to introduce *shifts* into the model. As defined in [Ima87], [SIM94], [Fle93][31] and [INC97][32], a shift is a set of time periods in which work is done. Outside of these time periods, no new inputs should happen in the model.

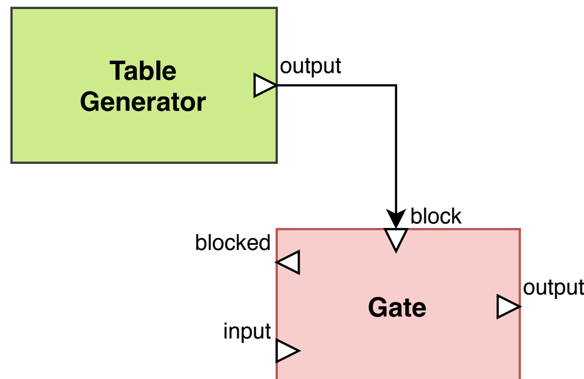



Figure 4.10: Assigning shifts to sections of the model.

[29] Or falsy.
[30] Or any other truthy value.
[31] In [Fle93], you can use time tables and preemption to model this.
[32] In [INC97], it can be found under the *AVAILABILITY* node of the atom tree.

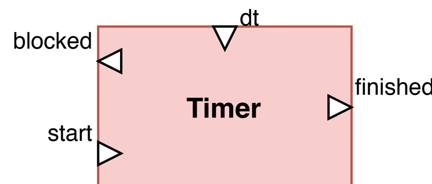
*Figure 4.10* shows a graphical example on how the shifts may be assigned, using a TABLE GENERATOR and a GATE. The former holds a schematic of the shift periods and the latter makes sure the messages won't enter the system outside of this schedule. For instance, a bank is open from 7.30 a.m. until 5 p.m. No customers should enter the bank outside of these office hours. Additionally, you could say the bank is always open, but the employees only work in those hours. This is a clear example on how shifts might work in a model.

### 4.7.8 Time Manipulation

No, this section is not going to go into detail on time travel and time dilation, but rather on building blocks that make use of the simulation time to handle certain events. Think of the *Wait Time* block from [Ima87] or the *Timer* from [LEG13].

#### Timer

The simplest time manipulation is to hold a message for a predefined delay $dt$. The building block that can do this is the TIMER , as shown in *block 4.37*.



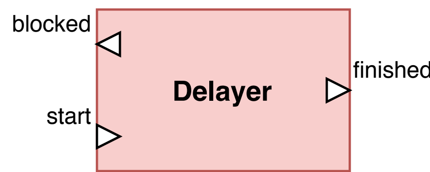

Block 4.37: TIMER building block.

When a message arrives on the *start* input, the TIMER will keep this item for $dt$ time, before releasing it on the *finished* output. All items that arrive on the *start* input while an item is being held are immediately bounced back on the *blocked* output port. If you do not want the items to be bounced back, you can make use of the QUEUE block with a predefined $dr$ attribute. Alternatively, the ADVANCE block may be used as well.

Note that each input event in DEVS recomputes the $ta$, meaning that $ta$ cannot simply be $dt$, but rather $dt - t_{elapsed}$. The $dt$ value can be altered during execution, based upon the input on the $dt$ port. If this value is less than the time that has been passed already, *finished* will be triggered at once.

#### Delayer

Not to be confused with the TIMER, the DELAYER (see *block 4.38*) omits the usage of the $dt$ attribute and the $dt$ input. Instead, it takes a function $f_t(i)$ that, for each item $i$ that arrives on the input *start*, will compute the time delay associated with this message.




Block 4.38: DELAYER building block.

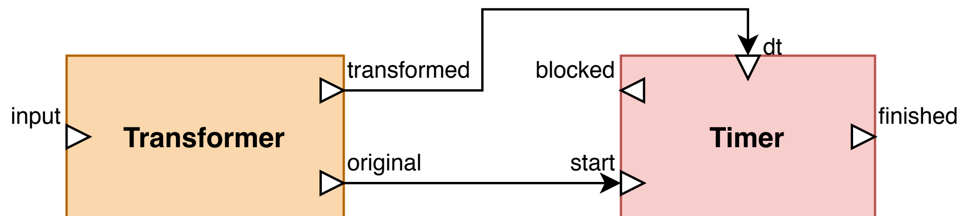



Figure 4.11: Creating the DELAYER building block from the TRANSFORMER and the TIMER.

*Figure 4.11* represents how the DELAYER can be seen w.r.t. the TIMER and the TRANSFORMER. While it can be implemented this way, it is much more efficient to use an Atomic DEVS instead.

### 4.7.9 Syncing

In *section 2.2.7*, it was mentioned that, because we're using CDEVS, it is perfectly possible that not all external events will arrive in the same timeframe. This is the reason why some building blocks (like the COUNTER and the TRANSFORMER) will output the input as well as the result of their implementation in the same output function. Unfortunately, this does not solve all our problems.

In *figure 4.12*, you can find a small model that allows us to test the validity of the GUARD building block. The CONSTANT GENERATOR continuously outputs a certain value, which is collected in the first TABLE COLLECTOR (`collector 1`) if there were no resources left. If there were, the items are enqueued in the SIMPLE QUEUE. This latter block allows each item to be in the critical section for a predefined time delay (hence the reneging). When an item leaves the critical section, it is collected in the second TABLE COLLECTOR (`collector 2`).

Now, what happens if an item leaves the critical section (that has no resources left) at the same time as another wants access? For instance, our GUARD has 3 resources available and our SIMPLE QUEUE still holds two of them. The logic defined for the GUARD block (see *section 4.7.6*) clearly states that the leave event should happen first, allowing the new item access to the SIMPLE QUEUE.

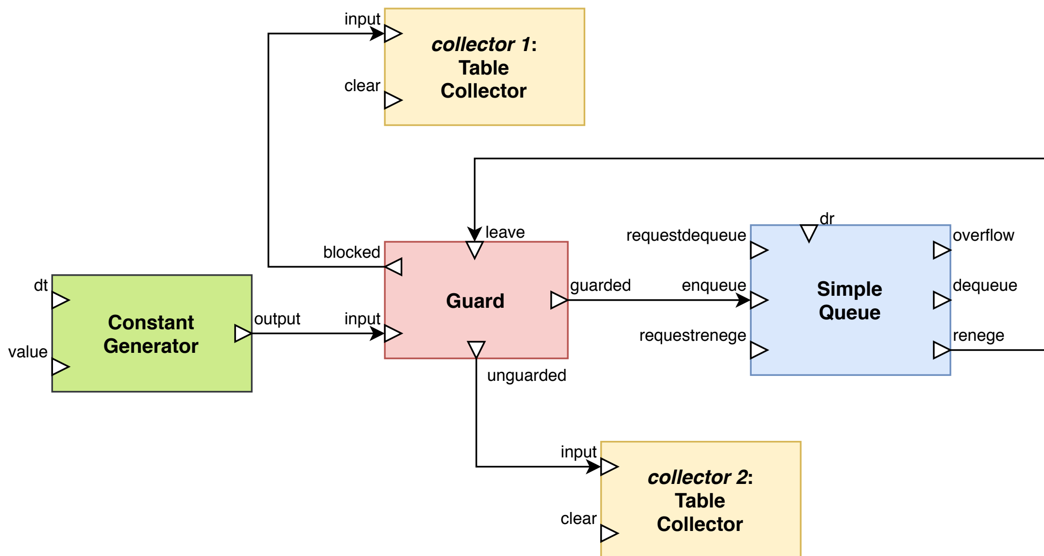
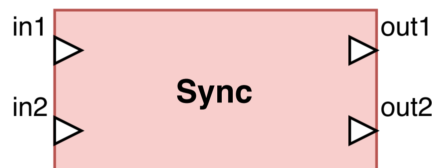



Figure 4.12: Issue with messages arriving at a different time.

Unfortunately, because the message on the *leave* port of the GUARD arrives a few timeframes after the message that the CONSTANT GENERATOR generates, the new frame will still be blocked in this structure.

We can make use of a TIMER block (see *section 4.7.8*) with a *dt* of 0 between the CONSTANT GENERATOR and the GUARD to fix the issue in this case. It solves the issue by exploiting the fact that the default *select* function in Python(P)DEVS executes the Atomic models in an alphabetical order. Additionally, all unnamed blocks in the figure were given the name of their type[33].

The model is small, hence, the problems will become more provocative the bigger our model gets, hence we follow [Fle93] and introduce the SYNC block (see *block 4.39*).




Block 4.39: SYNC building block.

The SYNC block has $n$ input and output ports and allows for syncing the arriving messages. Whenever all input ports have received a message, the block will release all these messages over the corresponding output ports. The

[33]Which is a very ugly and specific solution to a generic problem.

block does not enqueue the arriving items. A construction similar to *figure 4.9* is required to circumvent messages arriving too fast after one another.

## 4.8 Simulation Tracers

Often, it is a good idea to keep track of events that happen in the simulation. These can be especially useful when trying to gather statistics or obtain cost information ([Ima87], [SIM94]). Luckily, this can be done by using a simulation tracer.

A simulation tracer is not a building block, but rather an object that gathers the real time information of a simulation and processes it. Accompanied with PythonDEVS-BBL, there are four generic tracers[34], based on the functionality provided by the tools from *section 3.1*. Some examples on how to use each of these tracers are given in the Jupyter notebooks that can be found in `src/notebook`.

### 4.8.1 Plot Tracer

The `PlotTracer` provides a way of plotting your simulation data during runtime, given a set of collectors and a way of plotting. For instance, a Table Collector (*section 4.3*, *block 4.12*) can be plot easily in a line plot, but you might require items to be grouped together in buckets, in order to obtain valid data for a bar plot or histogram, or maybe you want to plot boxplots instead. All this info can be passed to the `PlotTracer` to plot the correct info.

#### A note on Boxplots

A boxplot is a box-and-whiskers plot, where the whiskers define the data range, the box defines the inter-quartile range $IQR = Q3 - Q1$ and the median is defined by a line in the box. Optional outliers are drawn as dots outside of the range. While the task of "drawing a boxplot" seems simple enough, but the issue lies in the fact that there is no single formula for computing $Q1$ and $Q3$. The six most common formulas are given below (see also [Lan06; FP87]). Assuming that we have a list $L$ of $N$ items, $Q1$ is the item at index $q1$ if $q1$ is an integer, otherwise it's the linear interpolation of the surrounding points. The same is true for $Q3$ and $q3$. Note that both $q1$ and $q3$ are one-based.

*Default Formula* The $Q1$ and $Q3$ values are the medians of the lower list and the upper list of the set split on its global median, respectively. The global median is not included in the computation. When there is an

[34] As well as a superclass for creating your own tracers.

even amount of items, the two values that make up the median are both excluded from the computation.

$$q1 = \begin{cases} \frac{N}{4} & \text{if } N \text{ is even} \\ \frac{N+1}{4} & \text{if } N \text{ is odd} \end{cases} \qquad q3 = \begin{cases} \frac{3 \cdot N + 4}{4} & \text{if } N \text{ is even} \\ \frac{3 \cdot N + 3}{4} & \text{if } N \text{ is odd} \end{cases} \tag{4.2}$$

*Tukey's Formula* [Tuk77] The formula that's used in R's [R F97] `fivenum` function and by the Python library Matplotlib [Dro+03] for plotting boxplots. The global median is included in the computation.

$$q1 = \begin{cases} \frac{N+2}{4} & \text{if } N \text{ is even} \\ \frac{N+3}{4} & \text{if } N \text{ is odd} \end{cases} \qquad q3 = \begin{cases} \frac{3 \cdot N + 2}{4} & \text{if } N \text{ is even} \\ \frac{3 \cdot N + 1}{4} & \text{if } N \text{ is odd} \end{cases} \tag{4.3}$$

*Minitab Formula* The formula that is used in Minitab [Min72] and in Microsoft Excel [Mic87] for the `QUARTILE.EXC` method (if you have a more recent version than 2007). The global median is excluded from the computation when the amount of items is odd.

$$q1 = \frac{N+1}{4} \qquad q3 = \frac{3 \cdot N + 3}{4} \tag{4.4}$$

*Freund & Perles' Formula* [FP87] This formula is used in Microsoft Excel for `QUARTILE.INC` (of for `QUARTILE`, if your version is less recent than 2007). Additionally, the R software makes use of this formula in the `summary` function. The Python library Numpy [Oli+95] also uses this in the `percentile` function.

$$q1 = \frac{N+3}{4} \qquad q3 = \frac{3 \cdot N + 1}{4} \tag{4.5}$$

*Moore & McCabe's Formula* [MM03] An alternative form of *formula 4.3*, where the global median is excluded from the computation. It is used in the TI-83 calculators for computing boxplot data [Lan06].

$$q1 = \begin{cases} \frac{N+2}{4} & \text{if } N \text{ is even} \\ \frac{N+1}{4} & \text{if } N \text{ is odd} \end{cases} \qquad q3 = \begin{cases} \frac{3 \cdot N + 2}{4} & \text{if } N \text{ is even} \\ \frac{3 \cdot N + 3}{4} & \text{if } N \text{ is odd} \end{cases} \tag{4.6}$$

*Mendenhall & Sincich's Formula* The same as formula 4.4, except with rounding. Usually, for the lower quartile, we round up and for the upper quartile, we round down. Other authors use other interpretations [Lan06].

$$q1 = \left\lceil \frac{N+1}{4} \right\rceil \qquad q3 = \left\lfloor \frac{3 \cdot N + 3}{4} \right\rfloor \tag{4.7}$$

Additionally, there is also the Fast Algorithm for Median Estimation (FAME, [FS07]), which estimates the median of a series, which can obviously be used to estimate $Q1$ and $Q3$ as well.

Nevertheless, all these algorithms require the full sequence to estimate the data, which we don't always have. In [JC85], an algorithm is created to estimate the percentiles, without the knowledge of the full sequence. See also the ESTIMATE COLLECTOR (*section 4.3*, *block 4.14*) for more info on this algorithm.

### 4.8.2 Statistics Tracer

Almost all tools that were discussed in *section 3.1* provide some way to obtain statistics about the throughput of a submodel (mostly of single blocks). They generally report the following statistics:

1. The amount of items that entered the submodel.
2. The amount of items that left the submodel.
3. The average time spent in the submodel.
4. The average amount of items in the submodel for a given period.

The `StatisticsTracer` provides exactly the same functionality, given a set of entry and exit points of a submodel.

### 4.8.3 Footprint Tracer

In [SIM94], the term "*footprint*" is used to indicate a consequence of using a building block. Other tools refer to this as a "*cost*"[35]. However you call it, a footprint is the cost of using a block (or a submodel). This can be a financial cost (when trying to compute how expensive your model is), or an environmental cost (when you, for instance, try to estimate the carbon footprint of your model).

Whenever the block is used, the cost of this block is accumulated until the end of the simulation. This can be done with the `FootprintTracer`. It takes a mapping from a port to a number and computes for each block the

[35] [SIM94] mainly uses "footprint" for the environment and "cost" for anything else.

associated cost. It is up to the user to determine what this cost actually means. Alternatively, the mapping can also map to a function that uses the item that enters the block to obtain a certain cost. For instance, an item that weighs 60 kilograms might be more costly to transport in comparison to an item that weighs 10 kilograms. Finally, this block will also allow an overview of the cost per item that traveled through the simulation. This allows users to quickly identify the most costly item in their system.

### 4.8.4 Profile Tracer

Finally, the last tracer, the `ProfileTracer` is a class that helps users create a profile of a simulation. It collects the amount of messages that traveled over certain connections and the number of times a block's internal, external and confluent[36] transition function fired.

While no tools provide such a feature, this might be used to optimize a spaghetti-like model. The resulting file is a CSV file, containing two tables (separated with an empty line): an overview for each block and an summary for each connection. Note that for each output that connects to multiple inputs, the connections will be count as well in the connection total.

## 4.9 Example

In the Pythin(P)DEVS documentation, there are lots of example usages of different parts of the framework. Additionally, there is a complete example use case, which can be found on https://msdl.uantwerpen.be/documentation/PythonPDEVS/queueing.html:

**Example 4.9.1** (Application to Queueing Systems).
*In this example we model the behaviour of a simple queue that gets served by multiple processors. Implementations of this queueing systems are widespread, such as for example at airport security. Our model is parameterizable in several ways: we can define the random distribution used for event generation times and event size, the number of processors, performance of each individual processor, and the scheduling policy of the queue when selecting a processor. Clearly, it is easier to implement this, and all its variants, in* DEVS *than it is to model mathematically. For our performance analysis, we show the influence of the number of processors (e.g., metal detectors) on the average and maximal queueing time of jobs (e.g., travellers). A model of this is shown in figure 4.13.*

*Events (people) are generated by a generator using some distribution function. They enter the queue, which decides the processor that they will be sent*

[36]Making it useful for parallel simulations, even if the library isn't.

*to. If multiple processors are available, it picks the processor that has been idle for the longest; if no processors are available, the event is queued until a processor becomes available. The queue works First-In-First-Out (FIFO) in case multiple events are queueing. For a processor to signal that it is available, it needs to signal the queue. The queue keeps track of available processors. When an event arrives at a processor, it is processed for some time, depending on the size of the event and the performance characteristics of the processor. After processing, the processor signals the queue and sends out the event that was being processed.*

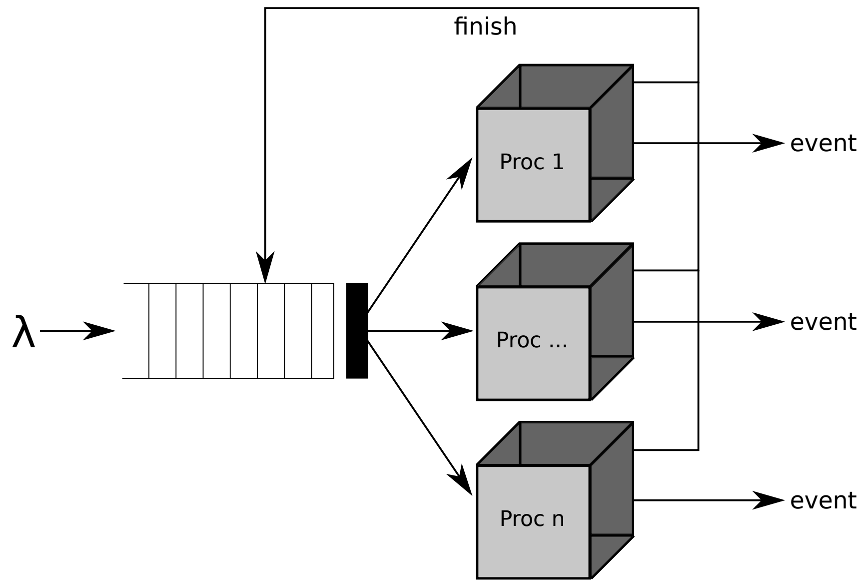



Figure 4.13: Application to Queueing Systems example model concept [Van14; VV17b].

Whereas the Python(P)DEVS documentation provides an in-depth description on how to implement this, we can, in fact, make us of our BBL. *Figure 4.14* is a graphical representation of the block diagram for three processors that produces the same results as the example solution. The implementation is given in a Jupyter notebook in the PythonDEVS-BBL repository.

Notice that, while the BBL has builtin RNGs, we bypass this functionality to enforce that the same distributions and seeds were used in the sample solution.

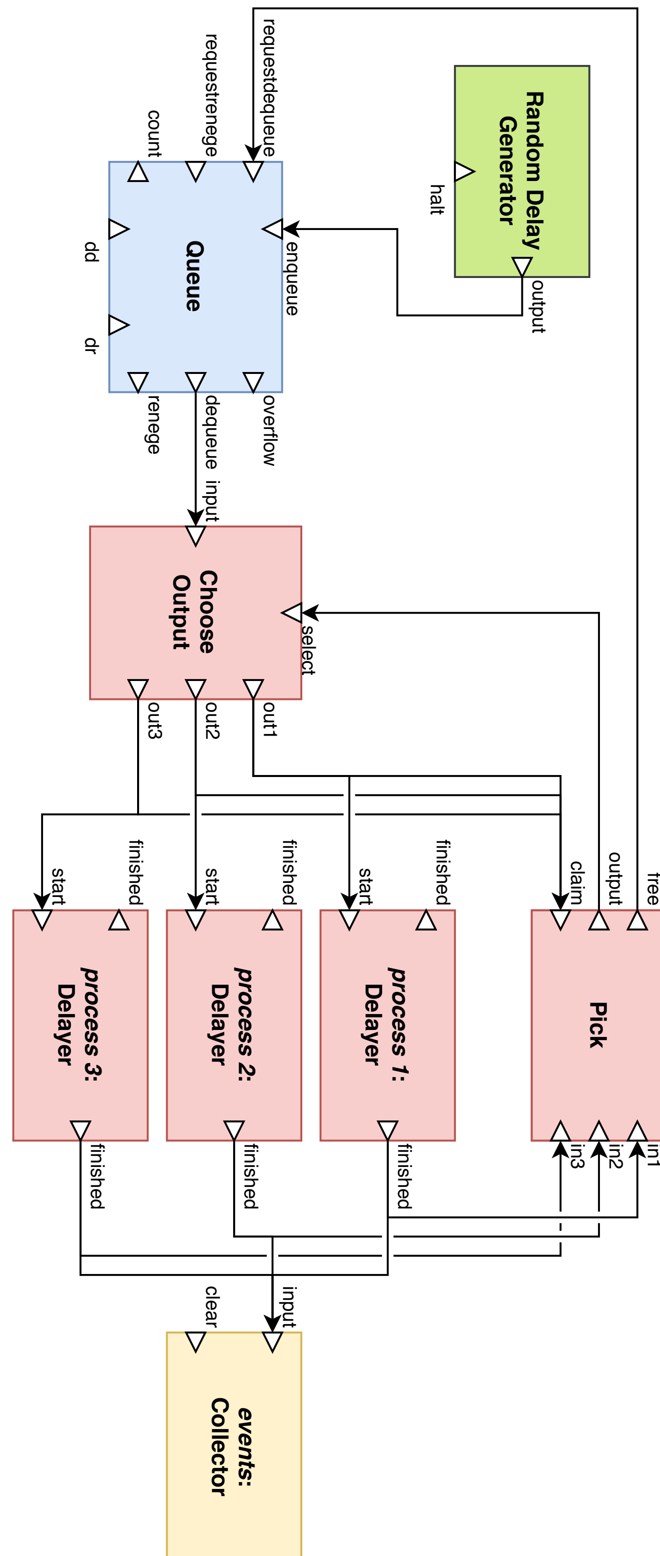



Figure 4.14: Implementation of *example 4.9.1* in PythonDEVS-BBL, for three processors.

## 4.10 Implementation Notes

This final section will go into detail on some additional remarks that are worth mentioning w.r.t. the library.

As should be clear by now, PythonDEVS-BBL is built in Python, on top of the Python(P)DEVS framework. As of 2020 (which is the year of writing), Python 2 has officially been discontinued by the Python Software Foundation. For this reason, PythonDEVS-BBL has not been tested on Python 2 and will, presumably, yield some strange behaviour when running it as such.

### 4.10.1 Library Specifics

While the library has been explained in much details, there are some aspects that need to be kept in mind.

**Constructor Arguments**

All building blocks will have a first (required) argument that represents the block name. While the `pypdevs.DEVS` class defaults this to be `None`, an error will be thrown if the name is not a string. It stands to reason it would be better to enforce the usage of the argument instead.

**Port Names**

The names for the ports in the code are exactly the same as described in this paper and the referencing variables follow the same names. I.e. `ConstantGenerator("gen").output` will obtain the *output* port (called "`output`") from a CONSTANT GENERATOR (which is called "`gen`").

Exceptions for this rule are the CHOOSE INPUT (see *section 4.7.5*, *block 4.33a*), CHOOSE OUTPUT (see *section 4.7.5*, *block 4.33b*), PICK (see *section 4.7.5*, *block 4.34*), SYNC (see *section 4.7.9*, *block 4.39*), ADDER (see *section 4.4.2*, *block 4.16*) and MULTIPLIER (see *section 4.4.2*, *block 4.17*) blocks, where the `inputs` and `outputs` members are lists that can be indexed and the corresponding port name will be `input-N` or `output-N`, as described in the documentation of these blocks.

**Project Structure**

PythonDEVS-BBL comes in three subpackages: `generic`, `extra` and `tracers`[37]. The first subpackage contains all the generic building blocks as described in this chapter, each category separated into their own module. The second one contains a set of modules, helper methods, classes and constructs for the building blocks to work. They also include the boxplot algorithms as described in *section 4.8.1*. Additionally, it provides a method `pypdevsbbl.extra.toDot`

[37] An additional subpackage, `domain`, is present to allow for future domain-specific blocks and the code for GPSS2DEVS, as will be discussed in *chapter 5*.

that creates a basic graph of any model, in a standard Graphviz [ATT91] format[38]. Lastly, the third subpackage contains a generalized basis for creating and assigning custom tracers. On top of that, it provides the tracers that are discussed in *section 4.8*.

### 4.10.2 Documentation

The PythonDEVS-BBL library comes with an extensive documentation that acts like an addendum to this thesis. When in doubt about the logic of a block, the *docstring* should probably provide an adequate answer. For each block, the documentation therefore provides:

- A short description of the block.
- An optional more in-depth description.
- A graphical representation of the block, as and when it is used within these pages.
- The constructor arguments. Including optional default values.
- A description of the state of the block, if it uses a state.
- The set of input ports for a block if they exist.
- The set of output ports for a block if they exist.
- Some optional notes and warnings which are good to know when using a block.

Non-blocks (i.e. functions, classes, variables...) will provide an sufficient description of what they are and how to use them.

**Jupyter Notebook**

Additionally, some examples and use cases have been implemented in Jupyter notebooks (see `src/notebook`). These provide some alternative explanations on the building blocks, specified on a combined usage in the use case.

Note that these use cases are fictional and only loosely based on real-life events and situations. They're there for illustrative and informative purposes and should not be used as a basis for research by themselves. In those contexts, a model should always be backed up with scientific results or experiments.

[38] Note that this is *not* a valid graphical representation w.r.t. the representations provided in this paper, but can be used to debug large models or validate the correctness thereof.

### 4.10.3 Tests

All code needs to be tested. This is an additional layer of safety for users to ensure that, with respect to the tests, the provided code act as it should and does not provide any unexpected behaviour. PythonDEVS-BBL is bundled with such a set of tests that can be executed as identified in the documentation. Rest assured that all required, important and described functionality is tested. Nevertheless, such a system is never foolproof and small issues or edge cases can slide through the cracks. Don't hesitate to contact the author when such issues occur.

The one exception to these tests is the SOUND block (see *section 4.5.3*, *block 4.25*). Because a unit test to check if the sound plays is quite useless, there is the possibility to execute the `pypdevsbbl.generic.io` module. As described in its documentation, you should hear a sound. If you don't, the test failed.

Also, do note that, while most tests run incredibly fast, this is not the case for the tests concerning the inverse distribution functions. This is because we require a lot of stochastic information to be able to deduct with enough certainty that our inverse cumulative distribution functions follow the correct distributions for a large numbers of parameters.

# CHAPTER 5

# GPSS2DEVS

In [VVV18; Van00b; Van00a], DEVS is said to be a common denominator for countless formalisms, even GPSS. Yet, no full explanation is given on this topic. Despite GPSS and DEVS being completely different world views, this chapter tries to provide a valid transformation from the GPSS subset we've discussed in *section 2.3* onto the CDEVS formalism. The interested reader is free to expand upon this transformation until it encompasses the full GPSS specification as is discussed in [Gor75].

While the generic idea is language-independent, we will focus on Python and Python(P)DEVS as we did in *chapter 4*. That being said, a graphical representation for most structures is provided in this paper, making it not only expandable to other frameworks, but also useful in visual specification languages for model transformations and their analysis [Gue+10; Gue+13; Hol97].

First, in *section 5.1*, we will discuss some of the choices we've made and preconditions that were set in order to obtain the transformation. *Section 5.2* goes into detail on the actual translational semantics for GPSS2DEVS. *Section 5.3* discusses some things to keep in mind when implementing the transformation and we end (in *section 5.4*) with a some examples that make use of the proposed transformation.

## 5.1 General Principles

Before we delve too deep into the actual transformation, we need to define some general principles we'll be using in this chapter. Additionally, *section 5.1.2* will describe some syntax we'll be using to prevent some ambiguity within the transformation.

### 5.1.1 Three Principles of GPSS2DEVS

We've set out to make GPSS2DEVS based on the following three principles:

1. Every GPSS building block needs to be transformed into a structure of DEVS building blocks in such a way that the original block can be recovered. I.e. every GPSS block has a representative DEVS alternative. In other words, the amount of blocks in the resulting DEVS model is *at least* the amount of blocks in the original GPSS model. The only exception to this rule is the `TRANSFER` block, which can be optimized as we'll discuss later on. We'll call this the *backwards propagation principle*.

2. Since only a single transaction may move at any time, we'll make it so there can be *at most* one transaction in each corresponding DEVS building block, simplifying the internal logic. This is the *limitation principle*, because it puts a constraint on the individual building blocks. An exception to this rule is obviously the `ADVANCE` block, which can contain multiple transactions at the same time.

3. The *availability principle* states that the GPSS chains need to be represented in the transformation in one way or another.

### 5.1.2 Notation

To prevent ambiguity, we need a way to distinguish the GPSS blocks from their DEVS counterpart. Especially when taking the *backwards propagation principle* (*principle 1*) quite literally. Hence, every DEVS building block that corresponds to a GPSS block (i.e. a GPSS2DEVS block) will be given the same name as it were in GPSS. Within this thesis, that means that the GPSS2DEVS blocks will be capitalized with the first letter only and they will be denoted with the Small Capitals font style (as was the case in *chapter 4*), while the GPSS counterparts will retain the `typewriter` style (as we did in *section 2.3*). For instance, `PREEMPT` becomes Preempt and `ASSIGN` becomes Assign.

Furthermore, arguments provided to GPSS building blocks will be denoted as class constructor parameters. I.e. "`GENERATE 5`" will translate to "Generate(5)".

## 5.2 Transformation

Now that we know the principles (as they were given in *section 5.1.1*) for a transformation, we can start mapping GPSS onto DEVS.

### 5.2.1 Entities

First things first, a transaction in GPSS can easily be seen as an event from DEVS, provided that an event is able to carry parameters. While a Python *dictionary* may seem like a valid representation of such a transaction, we'll use a class instead. This way, we have more flexibility over the transactions and allow for possible further expansions in a backwards-compatible way. Furthermore, the time-unit from GPSS can be mapped onto the time-unit from DEVS quite straightforwardly. We do need to keep in mind that real-time simulation in Python(P)DEVS indicates a time unit of a second.

The obtaining of SNAs can be done by making use of a global construct. Within the implementation, a function is used that takes the SNA as a string and returns the corresponding value, given the current transaction, the model and the current simulation time.

[Gor75] defines eight RNGs: `RN1` to `RN8`, where the number at the end defines the default seed. It also states that GPSS makes use of a MCG, which is quite a predictable generator in comparison to [LS03]. To make sure GPSS2DEVS is congruent to GPSS, we need to use an RNG that is as predictable[1]. Python's builtin `random` module was used for this purpose.

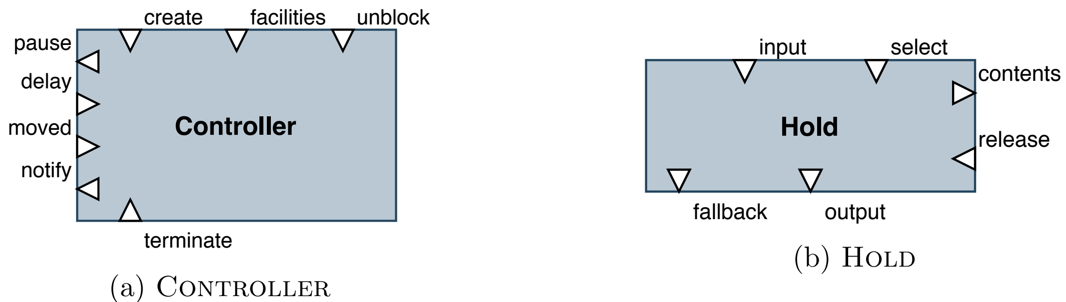
### 5.2.2 Scanning Algorithm

A valid transformation in GPSS2DEVS requires a control unit that coordinates the events being sent and, therefore, the flow of the transactions. We introduce a CONTROLLER block (see *block 5.1a*) and a HOLD block (see *block 5.1b*) that communicate how the transactions should move through the model. In effect, this communication implements the scanning algorithm. Whereas there is only one CONTROLLER block for each model, multiple HOLD blocks are apparent. In general, there is a single HOLD block *behind* every GPSS2DEVS block. This means that, in the general case[2], there are *at least two* DEVS blocks for every GPSS block.

HOLD blocks will listen to coordination messages that the CONTROLLER sends. Whenever a transaction enters such a block, it will be stored internally until the block is told by the CONTROLLER to *release* a specific transaction,

[1]This is mainly to ensure the validity when comparing simulation results.

[2]There are some exceptions to this rule.




(a) CONTROLLER

(b) HOLD

Block 5.1: GPSS2DEVS CONTROLLER and HOLD blocks.

which is done by a *notify* event. This internal storage makes it possible for the DEVS blocks to only look at a single transaction at any time, thus enforcing the *limitation principle* (principle 2).

There are two possible modes for every HOLD block: *send* mode, in which the contents of the HOLD block (i.e. all transactions that are waiting) are outputted over the *contents* port upon the arrival of a new *input* event; and *normal* mode, in which this does not happen. In our visual representation, we will say that whenever the *contents* port of a HOLD block $H$ is connected to the *moved* port of the CONTROLLER block, $H$ is in *send* mode.

*Principle 3*, the *availability principle* states that all chains need to be represented in GPSS2DEVS. Some of them are easily mapped, like the FEC, which is implicit in DEVS by making use of the *time advance* function *ta*. Others, like the CEC, are slightly more difficult to see[3]. We'll provide the CONTROLLER block with four internal lists:

*active* The list of transactions that are neither blocked, delayed, nor terminated. The scanning algorithm iterates over these transactions until the list is empty. The transactions on this list are ordered by priority (and time of arrival).

*delayed* The unordered list of transactions that are delayed until further notice. All transactions on the FEC can be found here, as well as transactions that enter a LINK/LINK block.

*created* The list containing all transactions that are created by GENERATE blocks during an iteration over the *active* list. We need to make sure all GENERATE blocks have called their output function $\lambda$ before the CONTROLLER starts with the scan. Additionally, all transactions that were delayed or blocked, but have moved, will be placed in this list, anticipating the rescan (hence, their $\lambda$ needs to have been called as well). The transactions do not need to be ordered in this list.

[3]The interrupt chain and user chains will be discussed later on.

*blocked* The list of transactions that cannot move to the next block due to some blocking condition. If a resource is released, the CONTROLLER will move all transactions on this list to the *created* list before rescanning. For performance reasons, the coordination of blocked transactions happens in groups instead of individually: if a single transaction of a group is blocked, the CONTROLLER can mark all remaining transactions in the group as blocked. The order of the transactions in this list does not need to be defined. Furthermore, the CEC can be seen as the union of the *active* and *blocked* lists.

Given these lists, we can rewrite the scanning algorithm that was given in *section 2.3.4* as follows:

1. The CONTROLLER will notify the first transaction $T$ in the *active* list that it may move. As long as $T$ can move through the model, it will move.
   - This is done by sending $T$ over the *notify* port and therefore notifying every HOLD block that $T$ is allowed to move.
   - In their turn, the HOLD blocks output $T$ over their output if it is waiting inside. The HOLD blocks remember $T$, until they receive another value from the CONTROLLER and immediately pass on $T$ if they receive the transaction on their *input*.
   - To prevent unnecessary overhead, the CONTROLLER won't send $T$ if the previous transaction that was allowed to move equals $T$.
2. Whenever $T$ is to be delayed or blocked, it will be removed from the *active* list and placed on the corresponding list. A delay happens when a transaction enters an ADVANCE[4] or a LINK block.
3. If $T$ leaves the FEC, the CONTROLLER is notified and adjusted accordingly. I.e. $T$ is moved from the *delayed* list to the *created* list, ready for the next iteration of the algorithm.
4. When $T$ is terminated, the termination counter of the CONTROLLER is updated as required by the TERMINATE block and $T$ is removed from the *active* list.
   - Because of the removal from this list, the next transaction in the *active* list will be selected for the next *notify* event automatically.
   - Whenever the termination counter reaches the zero, the DEVS simulation of the GPSS model is halted.

[4] The block is underlined to distinguish it from the previously declared ADVANCE block (see *block 4.11*).

### 5.2.3 Time and Flow

*Figure 5.1* shows the transformation of the `GENERATE`, `ADVANCE` and `TERMINATE` blocks. As you can see, every transaction that is generated will be added to the CONTROLLER's *created* list (by entering on the corresponding input). They are terminated from the simulation by entering on the CONTROLLER's *terminate* port[5]. The CONTROLLER will *always* have a connection between its *notify* output and a *release* input of a HOLD block.

Furthermore, the ADVANCE signals the CONTROLLER that a transaction has moved (i.e. is not delayed anymore) and the block *before* the ADVANCE will tell the CONTROLLER to move the transaction to the *delay* list. The CONTROLLER may *pause* certain transactions in the ADVANCE block. This block is, in fact, a Coupled DEVS and can be created as shown in *figure 5.2*.

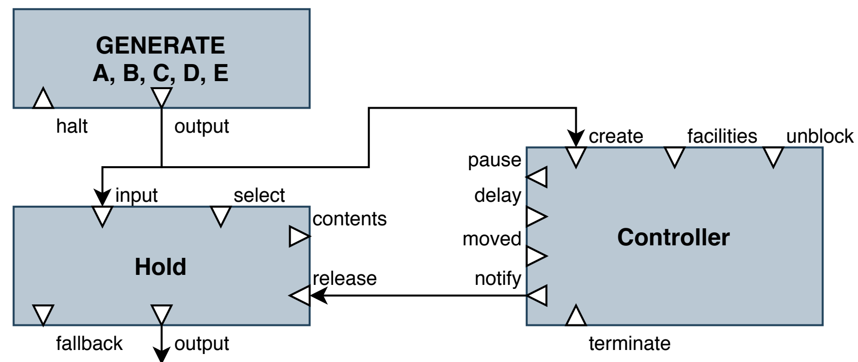
Seeing that we need to make all transactions unique and GPSS can have multiple `GENERATE` blocks, we need to transform the unique identifier that the GENERATE block[6] uses to initialize the transactions. If we don't, two GENERATE blocks will yield the same sequence of transactions, making them *not* unique. We can solve this issue by stating that the unique identifier (uid) of a transaction is given by $n \cdot i + g$, where $n$ identifies the amount of `GENERATE` blocks in the model, $g$ the index of the GENERATE block that creates the transaction and $i$ the amount of transactions the GENERATE block has created already. Now, each GENERATE block generates uid according to its own unique arithmetic progression, such that no to progressions can collide.

**Example 5.2.1** (Lucky Timings)**.**
*Consider the GPSS model with a visual concrete syntax as shown in figure 5.3a. The translated version of this model is straightforward, given figure 5.1. Starting from time 10, every 10 time units, a transaction is created with a uid that follows the sequence* $\{0, 1, 2...\}$*. Each transaction waits for 5 time units before being destroyed. The simulation runs until the termination counter becomes zero (i.e., until a given number of messages was destroyed). The sequence diagram given in figure 5.3b shows all events that are being sent in the translated example at time 10, 20, 30, ... In the diagram,* $Tx$ *identifies the transaction that is created in that time unit. The process goes through the following phases:*

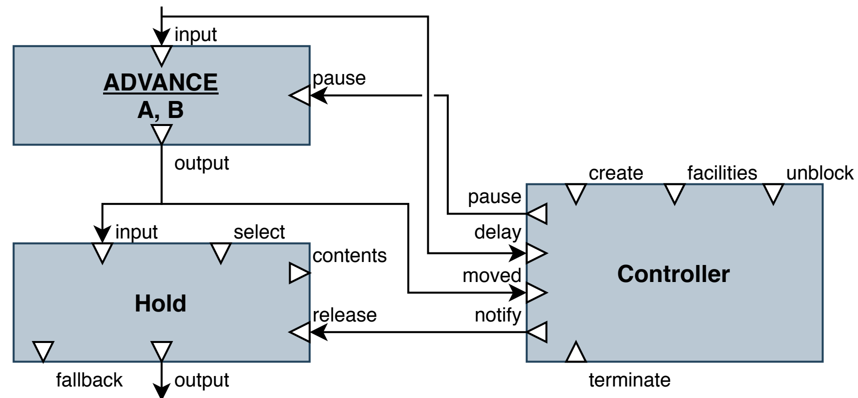
1. *When the* GENERATE *block creates* $Tx$*, it sends a message to the* CONTROLLER *stating that the transaction has been created. Next, it will pass the message on to a* HOLD *block.*

[5] Actually, a pair enters this port, where the first field indicates the to-terminate transaction and the second field the termination count for that transaction.

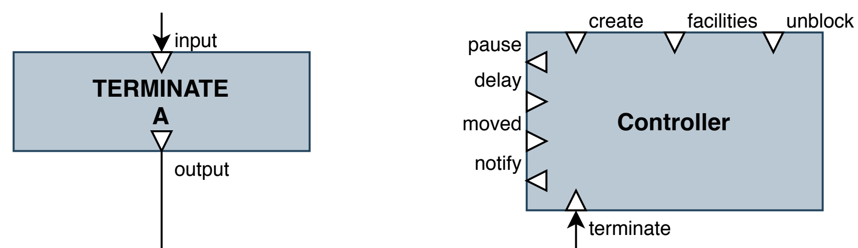
[6] Which is technically a Random Delay Generator (see *block 4.5*).




(a) GENERATE block in GPSS2DEVS.




(b) ADVANCE block in GPSS2DEVS.




(c) TERMINATE block in GPSS2DEVS.

Figure 5.1: The translation of the GENERATE, ADVANCE and TERMINATE blocks in GPSS2DEVS.

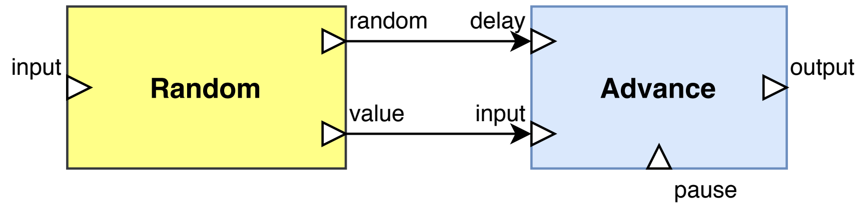



Figure 5.2: The GPSS2DEVS ADVANCE block as a Coupled DEVS.

2. *The* CONTROLLER *notifies all* HOLD *blocks that $Tx$ may move. Because $Tx$ is only in "`Hold 1`", it moves from there to the* ADVANCE *block, while also messaging the* CONTROLLER *that $Tx$ should be moved to the* delay *list, because it's entering an* ADVANCE*.*

3. *After five time units, $Tx$ is freed from the* ADVANCE *block and makes its way into the "`Hold 2`" block. The* CONTROLLER *is notified that $Tx$ is not delayed anymore and remembers that the last notification allowed $Tx$ to pass, so no notify is sent. "`Hold 2`" also remembers the previous notification and sends $Tx$ to the* TERMINATE *block.*

4. *The* TERMINATE *block makes the* CONTROLLER *remove $Tx$ from the active chain and reduce the termination counter by 1.*

5. *After 5 time units, the* GENERATE *block creates a new $Tx$, restarting the process.*

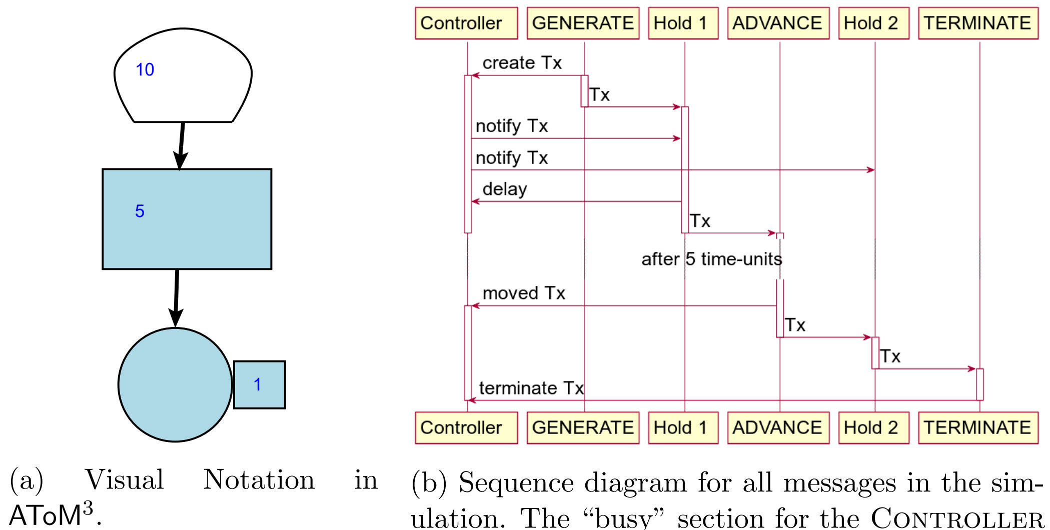
Let's defer the construction of the translation of the `TRANSFER` block until *section 5.2.5*. In the meantime, we can construct the translations for the `TEST` and the `ASSIGN` blocks without too much issues. They are shown in *figure 5.4*.

### 5.2.4 Resources

In general, resources follow the same structure as already described above. Each block that gains access (be it a SEIZE, a PREEMPT, an ENTER or a LOGIC) is immediately followed by a corresponding HOLD block. In fact, here, there is a double connection between the GPSS2DEVS-block and the HOLD block: one for the passing transaction and one that indicates if the transaction should be blocked[7].

For performance reasons, it's only these HOLD blocks that are in *send* mode and therefore notify the CONTROLLER about their contents. If the CONTROLLER tells the HOLD block to release a blocked transaction, the transaction retries to gain access on the GPSS2DEVS-block's *input* port[8].

[7] Using the *select* port of the HOLD block.
[8] Here, we use the *fallback* port of the HOLD block.




(a) Visual Notation in AToM[3].

(b) Sequence diagram for all messages in the simulation. The "busy" section for the CONTROLLER identifies that $Tx$ is located in the *active* list in these periods. For all other blocks, it indicates that $Tx$ is located in the block.

Figure 5.3: *Example 5.2.1.*

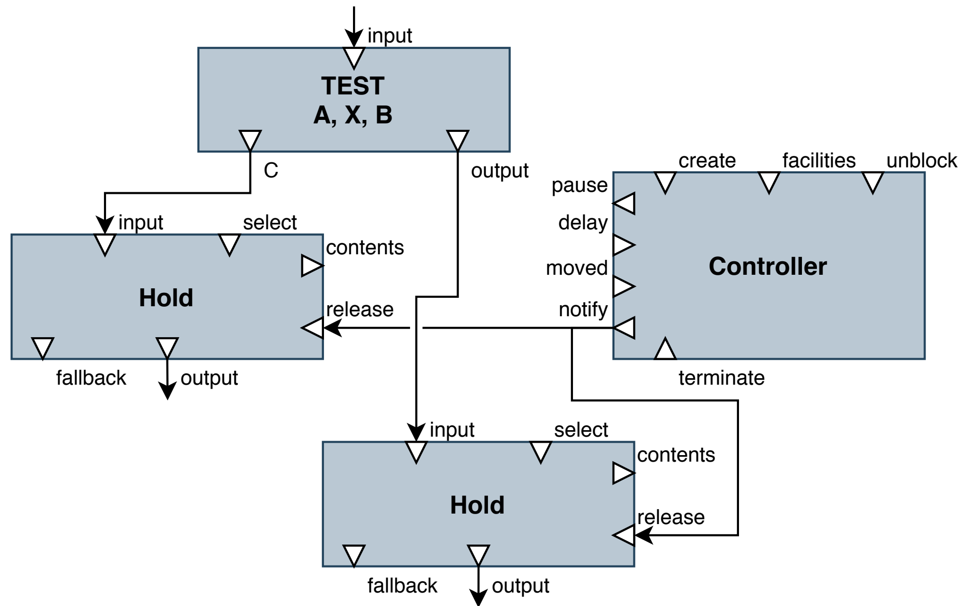
As soon as a retrying transaction is blocked, the CONTROLLER is aware that the resource cannot be accessed anymore and it marks all transactions that are in the same group as the trying transaction as "blocked"[9]. The block that requests access communicates with the corresponding resource whether or not access may be granted. Additionally, blocks that free up a resource also do a similar communication, besides also triggering a rescan.

**Facilities**

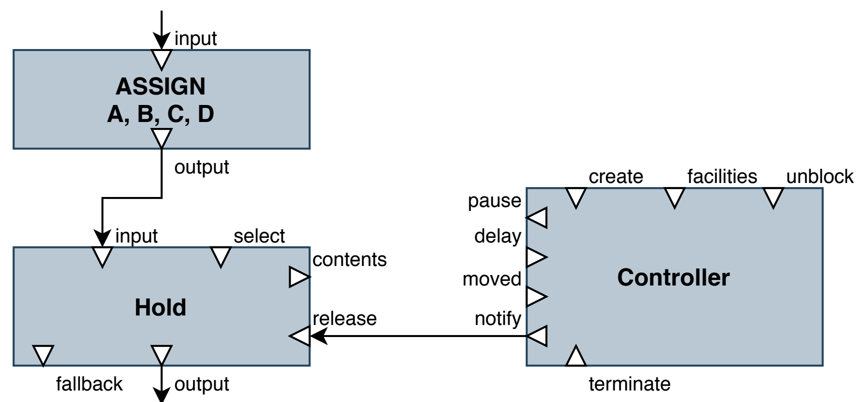
Besides the general rules that resources follow in the translation, facilities require some additional coordination with the CONTROLLER. In GPSS, a transaction has knowledge of all the facilities it belongs to. In the translation, this relationship is reversed: all facilities know which transactions have access to them, interrupted or not. This statement can be made without loss of generality, as long as we make sure a transaction may not be interrupted more than 255 times (as stated in [Gor75])[10]. To do so, we'll have the CONTROLLER listen to facility updates. When such an update happens, the CONTROLLER, using the *pause* port, must send a message to all ADVANCE blocks, notifying that all transactions (except the last transaction that obtained the facility) must be paused. Because of the way our transformation is constructed, a transaction can *only* be interrupted when its in an ADVANCE block. Only then, the list

[9]The HOLD blocks send their entire contents (i.e. the full group) to the CONTROLLER.

[10]While this may have been for memory reasons, this limit was kept in the translation, because that it is believed that such an edge case is indicative of bad modeling practices.

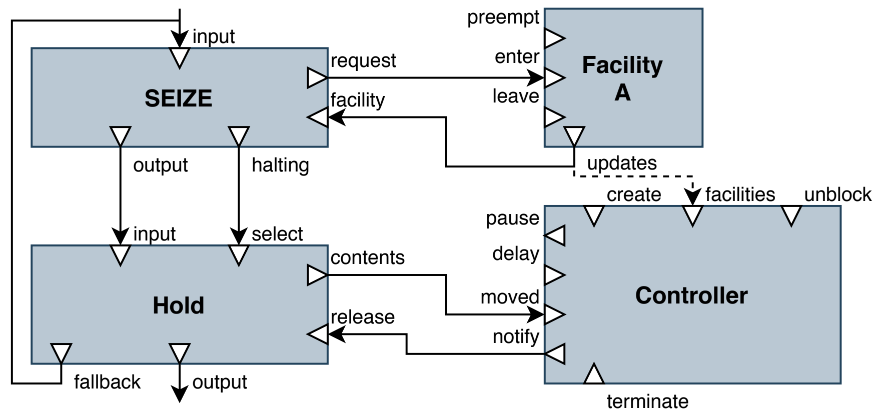



(a) TEST block in GPSS2DEVS.

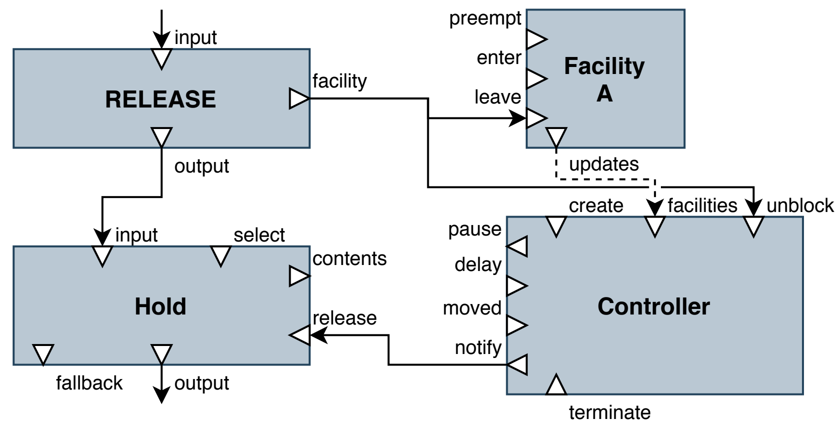



(b) ASSIGN block in GPSS2DEVS.

Figure 5.4: The translation of the TEST and ASSIGN blocks in GPSS2DEVS.




(a) SEIZE block in GPSS2DEVS.

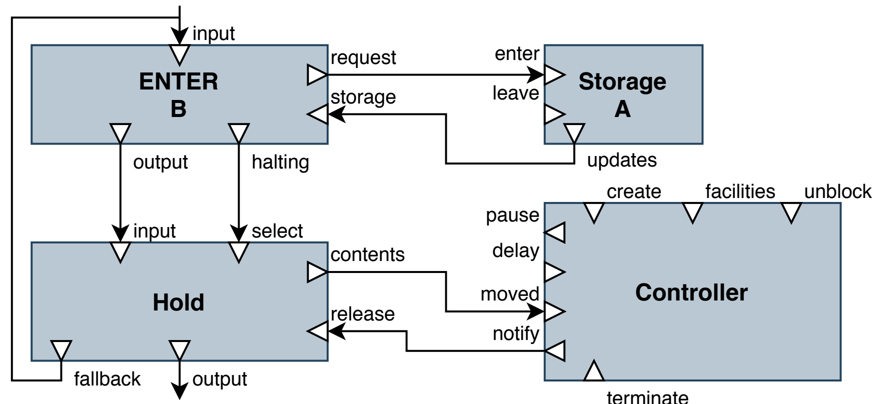



(b) RELEASE block in GPSS2DEVS.

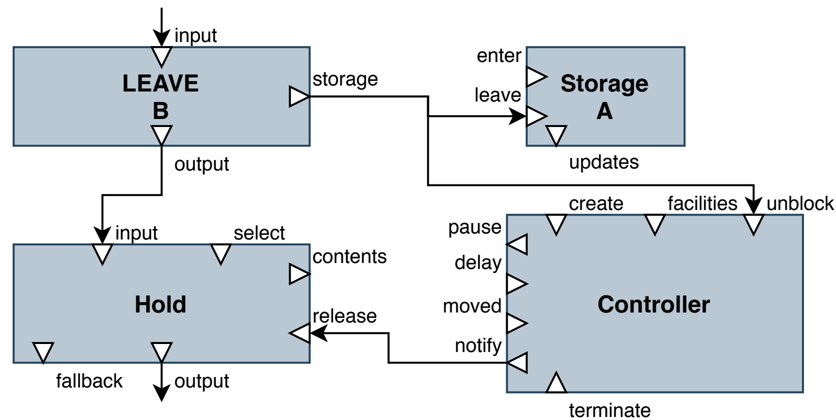
Figure 5.5: The translation of the SEIZE and RELEASE blocks in GPSS2DEVS.

of transactions that obtained the facility will be larger than 1. Whenever a transaction releases a facility (via either a RELEASE or a RETURN block), this list is updated and another transaction will be resumed.

*Figure 5.5* gives the translation for the SEIZE and RELEASE blocks. The dotted arrow indicates a connection that should be made upon the construction of the FACILITY. Notice how an additional block, FACILITY, is present in our model. This is done because the facilities can be obtained from any branch in the original model. Over the *updates* port, it outputs the current state of the facility, whenever some change happens. The translation for the PREEMPT and RETURN blocks can be inferred from this figure.




(a) ENTER block in GPSS2DEVS.




(b) LEAVE block in GPSS2DEVS.

Figure 5.6: The translation of the ENTER and LEAVE blocks in GPSS2DEVS.

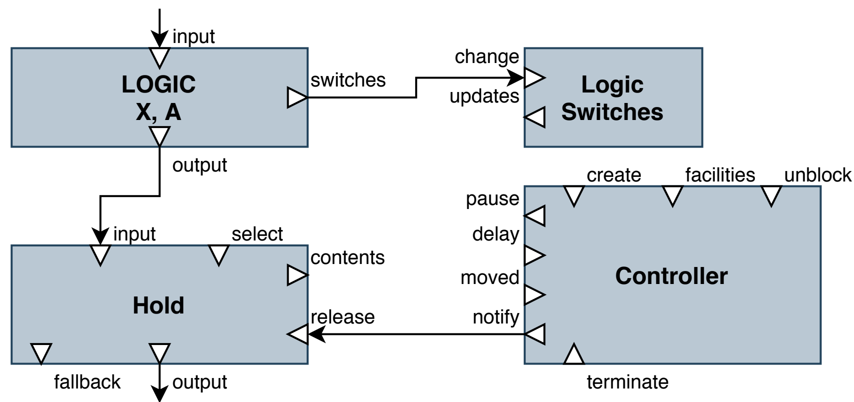
### Storages

Following the same structure as facilities, a STORAGE block can be added to the translation, identifying the storage the transactions are entering. In *figure 5.6*, the translation is shown.

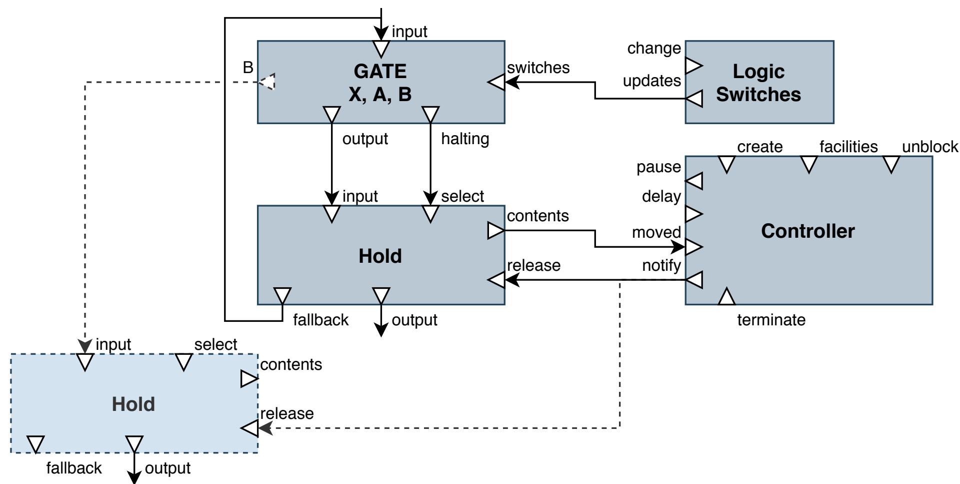
### Logic Switches

Logic switches are the odd ones out. Instead of having a single block for each switch, there is an all-knowing LOGICSWITCHES block. This is done because logic switches are more often than not accessed via SNAs. While we could manipulate Python's `__getattr__`-like functionality, this is a slow and rather dirty solution. Additionally, each logic switch only represents a boolean variable, so a unique block per switch would cause too much overhead.

Furthermore, instead of having a block that indicates a resource request and another that represents the release of that resource, logic switches only




(a) LOGIC block in GPSS2DEVS.




(b) GATE block in GPSS2DEVS.

Figure 5.7: The translation of the LOGIC and GATE blocks in GPSS2DEVS.

have a GATE block that can cause blocking. For this reason, the Gate block implements the same general logic as, for instance, an Enter block and a Leave block combined.

The translation of the LOGIC and the GATE blocks is given in *figure 5.7*. For the GATE, the optional fallback port B (when defined) also introduces an additional Hold block[11]. When argument B is not defined, the port is not created in the Gate block.

[11]Which is why it is shown in a muted color.

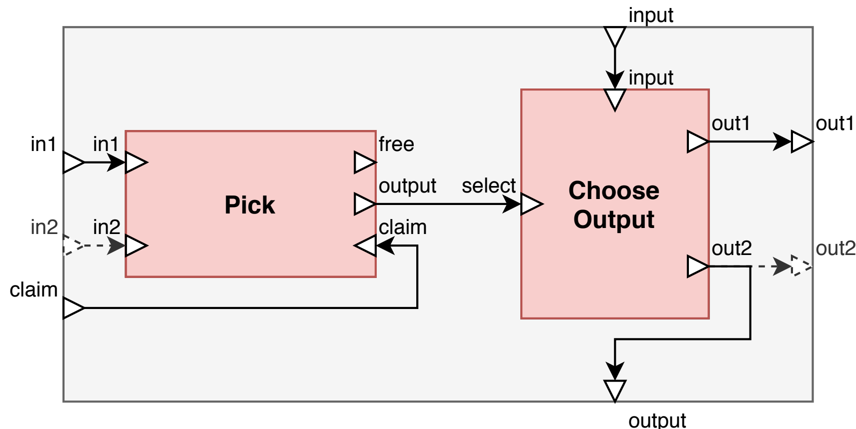



Figure 5.8: The GPSS2DEVS TRANSFER block in *conditional* mode as a Coupled DEVS.

## 5.2.5 The TRANSFER Block

As the exception that confirms the rule, the TRANSFER block is quite special: it does not follow the *backwards propagation principle* (*principle 1*).

In fact, the TRANSFER block in *unconditional* mode can be translated into a single connection. There is no need for a HOLD block here, seeing as the transaction will unconditionally flow from the previous block to the block indicated in parameter A. Blocking can never occur, so the addition of two blocks for this purpose would cause unnecessary overhead.

In the case of the *conditional* mode, for facilities and storages, the block can be translated as shown in *figure 5.9*. In the figure, the example for the combination with SEIZE and RELEASE is shown. The TRANSFER block follows the Coupled DEVS model that is represented in *figure 5.8*. Here, the *in2* and *out2* ports will not be used.

More generically, we can replace the TRANSFER block by a TEST block that checks the condition we're interested in. When we use a TRANSFER block in *all* mode, we can replace this by a chain of multiple TRANSFER blocks in *conditional* mode.

## 5.2.6 User Chains

As far as the translation is concerned, LINK and UNLINK can be seen as a special kind of ADVANCE block. The LINK enters transactions in the chain and the UNLINK takes them out again. The delay between entry and departure of the chain depends on the correspondence between process flows, hence all transactions that are on a chain will also be on the CONTROLLER's *delay* list. The translation for both blocks are shown in *figure 5.10*. Notice that the dotted lines indicate a possible additional part of the translation.

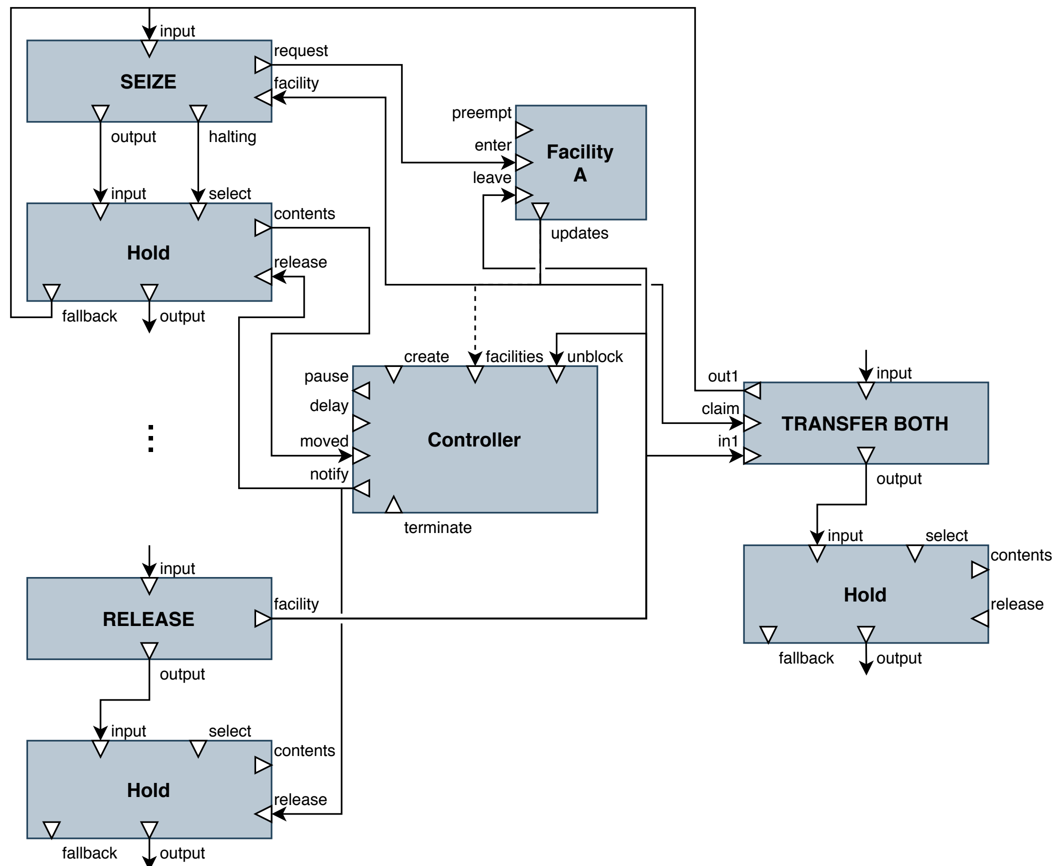
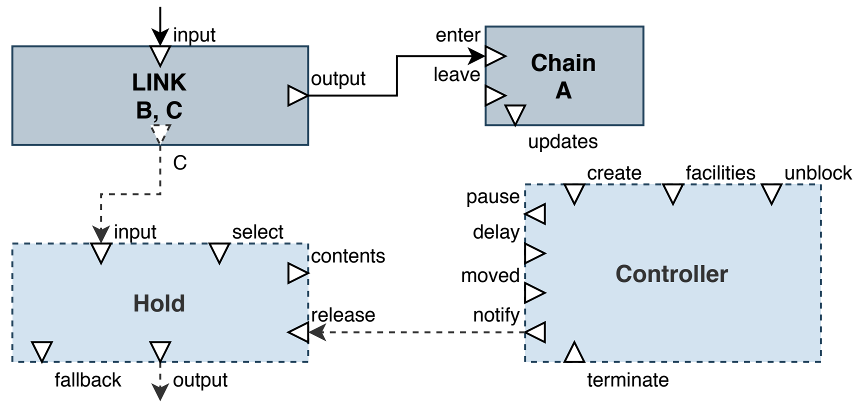



Figure 5.9: The translation of the TRANSFER block in *conditional* mode, combined with the SEIZE and RELEASE blocks in GPSS2DEVS.

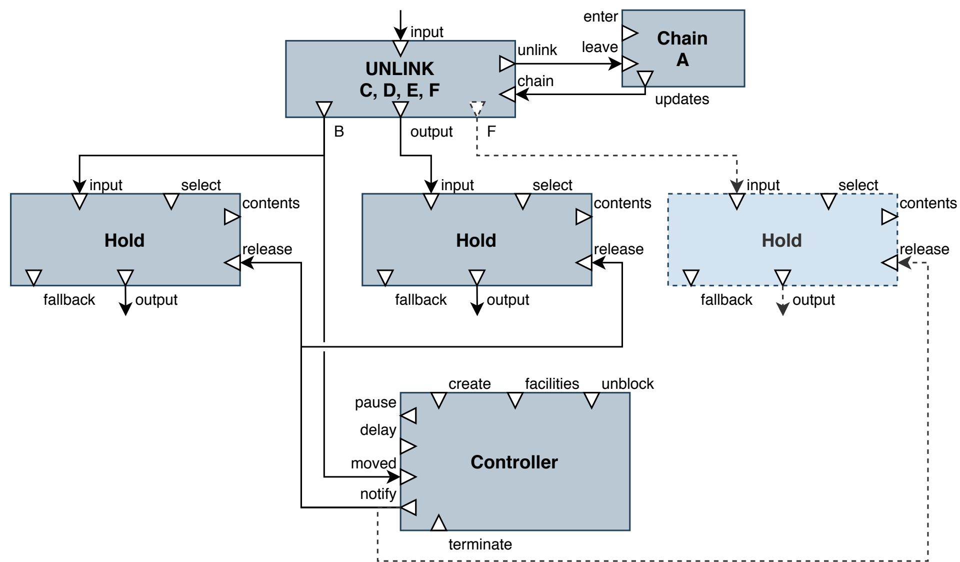
In the case of the LINK block, when C is set, the port will be connected to an existing HOLD block in the model. Notice that a HOLD block was chosen to keep it consistent, but (following the same reasoning as with the TRANSFER block in *unconditional* mode) we might as well refer to a GPSS2DEVS block. If C is undefined, the port won't be created. For the UNLINK block, when F is defined, we create an additional HOLD block for that port. Similar as to the LINK, when F is undefined, the port does not exist. Notice that the CHAIN block identifies a data structure that represents the user chain.

### 5.2.7 Gathering Statistics

As discussed in *section 2.3.9*, it is possible that statistics can be gathered during a GPSS simulation. Eventually, all statistics can be cleared from their corresponding blocks, similar to the RESET statement in GPSS.

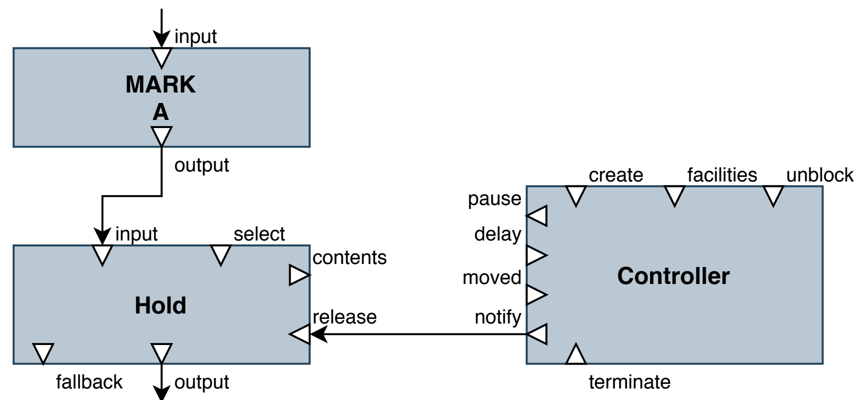



(a) LINK block in GPSS2DEVS.

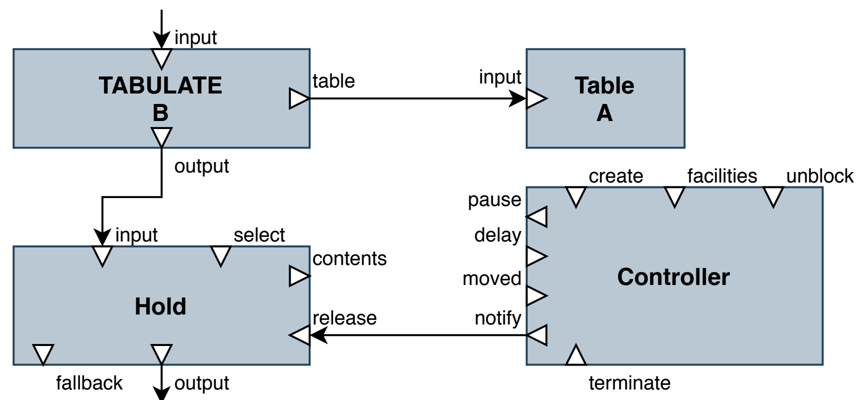



(b) UNLINK block in GPSS2DEVS.

Figure 5.10: The translation of the LINK and UNLINK blocks in GPSS2DEVS.




(a) `MARK` block in GPSS2DEVS.




(b) `TABULATE` block in GPSS2DEVS.

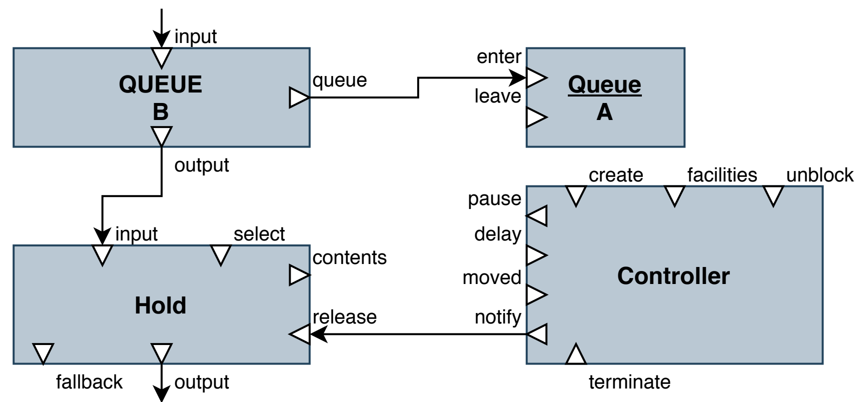
Figure 5.11: The translation of the `MARK` and `TABULATE` blocks in GPSS2DEVS.

### Tabulation

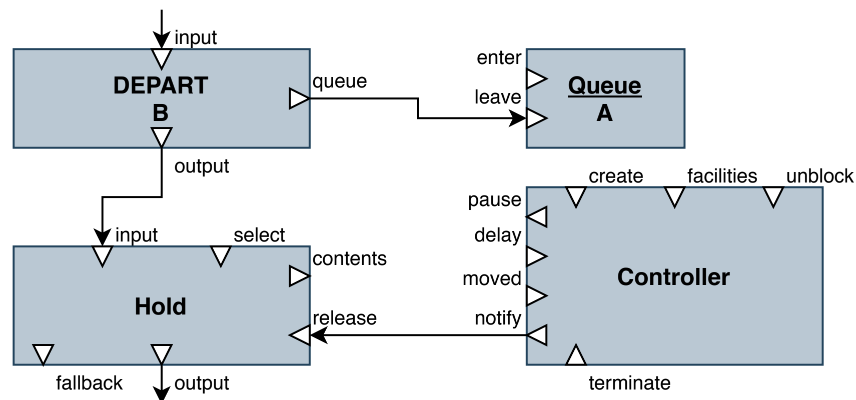
Let's add a DEVS block, namely a Table block, for each `TABLE` statement in the GPSS code. The Table's constructor sets up the buckets to be empty and a Tabulate block communicates to the Table what must be tracked. Additionally, we will compute the mean, variance and standard deviation in the same way as was discussed in *section 4.3.2*. The concerning GPSS blocks can be translated as is shown in *figure 5.11*.

### Queues

In a similar fashion as to the Table block, we can construct a Queue building block. It's underlined to distinguish from the translated Queue block that represents GPSS' `QUEUE`. Furthermore, we are *not* referring to the Queue block (i.e. *block 4.8b*) from *section 4.2*. *Figure 5.12* shows the corresponding translation.




(a) `QUEUE` block in GPSS2DEVS.




(b) `DEPART` block in GPSS2DEVS.

Figure 5.12: The translation of the `QUEUE` and `DEPART` blocks in GPSS2DEVS.

## 5.3 Implementation

While we have described the full translation of the subset of GPSS at this point, we still need to take a look at some implementation details concerning this translation.

### 5.3.1 Name Mangling

Notice how we're using CDEVS as a formalism. This makes it so events are executed within certain timeframes (see *section 2.2.7*). Therefore, we cannot assume that all events will fire within a certain order, as was hinted at in *section 5.2.2*. Luckily, the *select* function comes to save the day.

The order in which the events need to be executed is as follows:

1. All GENERATE blocks need to create the transactions for the current simulation time.
2. All ADVANCE blocks need to output the delayed transactions and notify the CONTROLLER that those transactions are no longer delayed. The same can be said for all UNLINK blocks.
3. The CONTROLLER needs to execute its scan.
4. All resources must communicate their state, so the corresponding blocks know what to do.
5. All HOLD blocks must release the requested active transaction to move.
6. The remainder of the blocks need to move the active transaction to the next HOLD block.

In order to encode this behaviour, we rename every building block in such a way that we can execute the blocks in alphabetical order[12]. To do this, we make use of *name mangling*. Name mangling is the process of creating unique names by encoding properties into the name. In this context, the names of the blocks can be made unique by encoding the line number if no label is given in the original GPSS. Furthermore, we encode the execution order by prefixing each block with `GPSS2DEVS_i_`, where `i` corresponds to the order that's listed above. In the GPSS2DEVS implementation, zero-based index of the given order was used.

### 5.3.2 Python(P)DEVS Formats

In the implementation of the proposed transformation, we can do two things:

1. For every GPSS block we find in the original model, we create a structure that corresponds to that block.
2. We provide Python *blueprints* for each structure in the form of a simple function call. We now have to translate the GPSS block to the corresponding function and provide the correct arguments.

Depending on the translation method, either possibility is valid. Yet, when using the latter option, there is less room for errors, because the blueprints are valid w.r.t. the translational semantics. A downside of using the blueprint-method is that the actual transformation is hidden and may be interpreted as a 1-to-1 mapping, which is not the case.

[12]This is the default order that's set by the *select* function in Python(P)DEVS.

## 5.4 Examples

Let's take a look at a complete example of the transformation, based on some GPSS examples that were introduced in [Gor78a]. Additionally, we will compare our results to GPSS World [Min10] and GPSS/H [Cra97].

### 5.4.1 Manufacturing Shop

In [Gor78a], an example of a simple manufacturing shop is provided, which will be reused to demonstrate the transformation on a small scale.

**Example 5.4.1** (Manufacturing Shop).
*A machine tool in a manufacturing shop is turning out parts at the rate of one every 5 minutes and places them on a conveyor that carries the parts towards three inspectors.*

*It takes 2 minutes to reach the first inspector; if he is free at the time the part arrives, he takes it for inspection. If he is busy at that time, the part takes a further 2 minutes to reach the second inspector, who takes the part if he is not busy. Parts that pass the second inspector may get picked up by the third inspector, who is a further 2 minutes along the conveyor belt; otherwise they are lost.*

*To keep the model small, only the transit time of the parts will be recorded and the possibility of the inspectors rejecting the parts will be ignored. Each inspector takes* $12 \pm 9$ *minutes per inspection.*

The model for this example is shown graphically in *figure 5.13* and textually in *figure 5.14*. The short version of the translated code is given in *figure 5.15*. An implementation in a Jupyter notebook is accompanied with the code. *Table 5.1* summarizes the results we obtain in comparison of the `TRANSIT` table and the facilities between GPSS World, GPSS/H and GPSS2DEVS. We used a warm-up of 10 parts and simulated the arrival of 10 000 parts. Given the results we obtain, we can conclude that the translation is equivalent to the original model in GPSS.

### 5.4.2 Telephone Exchange

On a larger scale, we have the telephone exchange example from [Gor78a]. This example includes almost everything that can be used by our subset of GPSS.

**Example 5.4.2** (Telephone Exchange).
*Assume we have a telephone system in which a series of calls come from a number of telephone lines and the system is to connect the calls by using one*

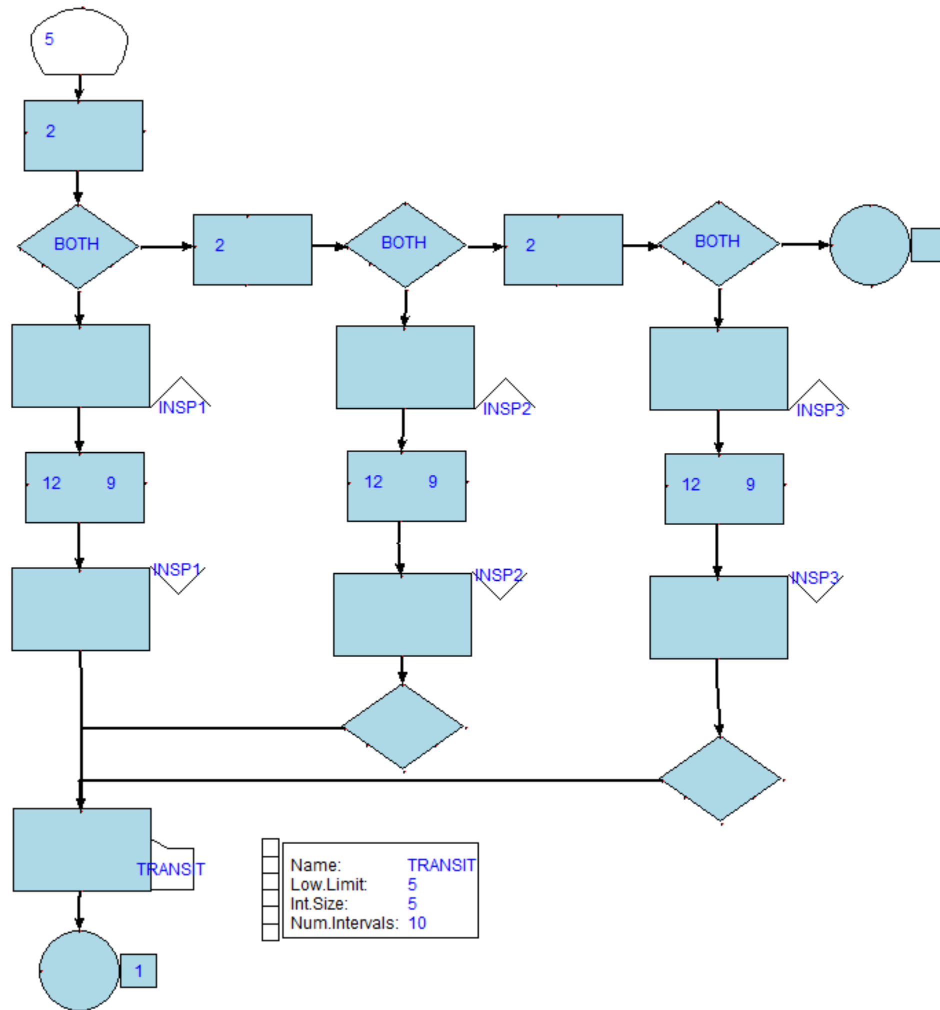



Figure 5.13: Visual Notation in AToM$^3$ for *Example 5.4.1* (the manufacturing shop).

| **Block** | **Metric** | **GPSS World** | **GPSS/H** | **GPSS2DEVS** |
|---|---|---|---|---|
| TRANSIT | Mean | 15.734 | 15.7752 | 15.6721 |
| | Standard Deviation | 5.479 | 5.4223 | 5.6909 |
| INSP1 | Utilization | 0.831 | 0.834 | 0.8603 |
| INSP2 | Utilization | 0.731 | 0.729 | 0.7419 |
| INSP3 | Utilization | 0.549 | 0.559 | 0.5382 |

Table 5.1: Obtained metrics for *example 5.4.1*. Ran until 10 000 parts accepted.

```
*                 Manufacturing shop model 5
*                 G. Gordon Figure 11-1/9-10
       SIMULATE
 L0    GENERATE   5                ; Create parts
 L7    ADVANCE    2                ; Move to the first inspector
 L8    TRANSFER   BOTH,L5,CONV1    ; Check if first inspector is busy
 L5    SEIZE      INSP1            ; The first inspector becomes busy
 L1    ADVANCE    12,9             ; Inspect
 L9    RELEASE    INSP1            ; Free inspector 1
 TAB   TABULATE   TRANSIT          ; Tabulate parts' transit time
 ACC   TERMINATE  1                ; Accepted parts
 CONV1 ADVANCE    2                ; Move to 2nd inspector
 C2    TRANSFER   BOTH,L13,CONV2   ; Check if 2nd inspector is busy
 L13   SEIZE      INSP2            ; The 2nd inspector becomes busy
 L15   ADVANCE    12,9             ; Inspect
 L17   RELEASE    INSP2            ; Free inspector 2
 L19   TRANSFER   ,TAB             ; To tabulate
 CONV2 ADVANCE    2                ; Move to 3rd inspector
 C3    TRANSFER   BOTH,L14,TERM    ; Check if 3rd inspector is busy
 L14   SEIZE      INSP3            ; The third inspector becomes busy
 L16   ADVANCE    12,9             ; Inspect
 L18   RELEASE    INSP3            ; Free inspector 3
 L20   TRANSFER   ,TAB             ; To tabulate
 TERM  TERMINATE                   ; Remain uninspected
 TRANSIT TABLE      M1,5,5,10
       START      10,NP
       RESET
       START      10000
       END
```

Figure 5.14: Textual Notation for *Example 5.4.1* (the manufacturing shop).

```
from pypdevsbbl.domain.gpss import GPSS2DEVS, dist

class Model(GPSS2DEVS):
    def __init__(self):
        super().__init__("Manufacturing")

        self.createFacility("INSP1")
        self.createFacility("INSP2")
        self.createFacility("INSP3")
        self.createTable("TRANSIT", "M1", 5, 5, 10)

        self.GENERATE("L0", dist=lambda x: 5)
        self.ADVANCE("L7", dist=lambda x: 2, prev="L0")
        self.SEIZE("L5", "INSP1")
        self.ADVANCE("L1", dist=dist, args=(12,9), prev="L5")
        self.RELEASE("L9", "INSP1")
        self.TABULATE("TAB", "TRANSIT")
        self.TERMINATE("ACC", '1')
        self.TRANSFER_BOTH("L8", "L5", "INSP1", self.GPSS2DEVS_5_L9.facility)
        self.ADVANCE("CONV1", dist=lambda x: 2, prev="L8")
        self.SEIZE("L13", "INSP2")
        self.ADVANCE("L15", dist=dist, args=(12,9), prev="L13")
        self.RELEASE("L17", "INSP2")
        self.TRANSFER_BOTH("C2", "L13", "INSP2", self.GPSS2DEVS_5_L17.facility)
        self.ADVANCE("CONV2", dist=lambda x: 2, prev="C2")
        self.SEIZE("L14", "INSP3")
        self.ADVANCE("L16", dist=dist, args=(12,9), prev="L14")
        self.RELEASE("L18", "INSP3")
        self.TRANSFER_BOTH("C3", "L14", "INSP3", self.GPSS2DEVS_5_L18.facility)
        self.TERMINATE("TERM")


        self.connectPorts(self.GPSS2DEVS_4_L0.output, self.GPSS2DEVS_1_L7.input)
        self.connectPorts(self.GPSS2DEVS_4_L7.output, self.GPSS2DEVS_5_L8.input)
        self.connectPorts(self.GPSS2DEVS_4_L5.output, self.GPSS2DEVS_1_L1.input)
        self.connectPorts(self.GPSS2DEVS_4_L1.output, self.GPSS2DEVS_5_L9.input)
        self.connectPorts(self.GPSS2DEVS_4_L9.output, self.GPSS2DEVS_5_TAB.input)
        self.connectPorts(self.GPSS2DEVS_4_TAB.output, self.GPSS2DEVS_5_ACC.input)
        self.connectPorts(self.GPSS2DEVS_4_L8.output, self.GPSS2DEVS_1_CONV1.input)
        self.connectPorts(self.GPSS2DEVS_4_CONV1.output, self.GPSS2DEVS_5_C2.input)
        self.connectPorts(self.GPSS2DEVS_4_L13.output, self.GPSS2DEVS_1_L15.input)
        self.connectPorts(self.GPSS2DEVS_4_L15.output, self.GPSS2DEVS_5_L17.input)
        self.connectPorts(self.GPSS2DEVS_4_L17.output, self.GPSS2DEVS_5_TAB.input)
        self.connectPorts(self.GPSS2DEVS_4_C2.output, self.GPSS2DEVS_1_CONV2.input)
        self.connectPorts(self.GPSS2DEVS_4_CONV2.output, self.GPSS2DEVS_5_C3.input)
        self.connectPorts(self.GPSS2DEVS_4_L14.output, self.GPSS2DEVS_1_L16.input)
        self.connectPorts(self.GPSS2DEVS_4_L16.output, self.GPSS2DEVS_5_L18.input)
        self.connectPorts(self.GPSS2DEVS_4_L18.output, self.GPSS2DEVS_5_TAB.input)
        self.connectPorts(self.GPSS2DEVS_4_C3.output, self.GPSS2DEVS_5_TERM.input)
```

Figure 5.15: Python(P)DEVS code for *Example 5.4.1* (the manufacturing shop).

*of a limited number of links. Only one call can be made to any one line at a time and it is assumed that the calls are lost if the called party is busy or no link is available.*

*It will be assumed that the distribution of arrivals is Poisson with a mean interarrival time of 12 seconds. The length of the calls will also be assumed to have an exponential distribution. It will be assumed that each new call can come from any of the non-busy lines with equal probability, and that its destination is equally likely to be any line other than itself.*

*In the telephone system, blocked class wait for a link to become free with the following service rules. Line 1 belongs to the company president. If there is an incoming call for line 1 and line 1 is free, the next free link goes to that call. Otherwise, the link goes to the call with the lowest origin number.*

The model for the example is given graphically in *figure 5.16* and textually in *figure 5.17*. The corresponding code can be found in *figure 5.18*. Similarly to the previous example, a sample implementation of the execution is given in a Jupyter notebook. *Table 5.2* summarizes the results for the `LNKS` storage and the `WAIT` chain for GPSS World, GPSS/H and GPSS2DEVS. The model cannot be made deterministic, while still remaining consistent with the use case.

It is interesting to note that the results we obtain lie closer to the GPSS/H results than the obtained values from GPSS World. This is especially the case when comparing the average time that each transaction spent in the chain.

| **Block** | **Metric** | **GPSS World** | **GPSS/H** | **GPSS2DEVS** |
|---|---|---|---|---|
| LNKS | Utilization | 0.683 | 0.728 | 0.7079 |
| | Average Contents | 6.834 | 7.277 | 7.0788 |
| WAIT | Average Time | 41.134 | 24.474 | 23.5716 |
| | Average Contents | 0.426 | 0.483 | 0.4045 |

Table 5.2: Obtained metrics for *example 5.4.2*. Ran for 10 hours.

Figure 5.16: Visual Notation in AToM$^3$ for *Example 5.4.2* (the telephone exchange).

```
*      SIMULATION OF A TELEPHONE SYSTEM - MODEL 2 (president line)
POISS  FUNCTION  RN1,C24              Function for I/A interval (Poisson)
0.0,0.0/0.1,0.104/0.2,0.222/0.3,0.355/0.4,0.509/0.5,0.69/
0.6,0.915/0.7,1.2/0.75,1.38/0.8,1.6/0.84,1.83/0.88,2.12/
0.9,2.3/0.92,2.52/0.94,2.81/0.95,2.99/0.96,3.2/0.97,3.5/
0.98,3.9/0.99,4.6/0.995,5.3/0.998,6.2/0.999,7/0.9997,8

*      The phone process

       GENERATE  12,FN$POISS,,,,2PH   ; Arrival of calls, characterized by two halfword parameters,
*                                       one for ORIGin and one for DESTination
       TEST G    V$FREEL,2,ABND       ; Test whether system is full (all lines in use)
ASN1   ASSIGN    ORIG,V$LN,PH         ; Pick an ORIGin LiNe, store in parameter ORIG
       GATE LR   PH$ORIG,ASN1         ; Check whether ORIGin line is busy; if so, go back and try
          ; to pick another line
ASN2   ASSIGN    DEST,V$LN,PH         ; Pick a DESTination LiNe, store in parameter DEST
       TEST NE   PH$ORIG,PH$DEST,ASN2 ; Retry if DESTination == ORIGin
       LOGIC S   PH$ORIG              ; Set ORIGin line busy
       TRANSFER  BOTH,,BLKD           ; If no link available, blocked in WAIT chain
GETL   ENTER     LNKS                 ; Get a link
       GATE LR   PH$DEST,BUSY         ; Check whether DESTination line is busy
       LOGIC S   PH$DEST              ; Set DESTination line busy
       ADVANCE   120,FN$POISS         ; Talk for a while, talktime mean 12min
       LOGIC R   PH$ORIG              ; ORIGin hangs up
       LOGIC R   PH$DEST              ; DESTination hangs up
       LEAVE     LNKS                 ; Free up the link
CKCH   TEST G    CH$WAIT,0,TERM       ; Test whether calls are waiting
       GATE LR   1,GETF               ; Check whether line 1 (president) is free
       UNLINK    WAIT,GETL,1,DEST$PH,1,GETF ; If a call to 1 (president) is waiting, get it out
*                                             of the WAIT chain and let it get a link, else connect
*                                             (in the other UNLINK block) the first
*                                             waiting call (ordered by call ORIGin)
TERM   TERMINATE                      ; Normal end of a succesful call
GETF   UNLINK    WAIT,GETL,1          ; Connect the first waiting call (ordered by call ORIGin)
       TRANSFER  ,TERM
ABND   TERMINATE                      ; Abandon call
BLKD   LINK      WAIT,ORIG$PH         ; Wait in order of call ORIGin (lower number first)
BUSY   LOGIC R   PH$ORIG              ; Caller hangs up
       LEAVE     LNKS                 ; Free up the link
       TRANSFER  ,CKCH                ; Go to test for waiting calls
LNKS   STORAGE   10                   ; Number of links
LN     VARIABLE  XH$NLINES*RN1/1000+1        ; Pick a random LiNe
FREEL  VARIABLE  XH$NLINES-2*S$LNKS-CH$WAIT  ; Number of free lines

*      The clock process

       GENERATE  60            ; One clock tick every minute
       TERMINATE 1

*      GPSS/H Control Statements

       INITIAL   XH$NLINES,50  Set the total number of lines
       START     10,NP         Warm up run of 10 minutes, don't print
       RESET                   Wipe out statistics
       START     600           Main run (10 hours)

       END
```

Figure 5.17: Textual Notation for *Example 5.4.2* (the telephone exchange).

```
from pypdevsbbl.domain.gpss import GPSS2DEVS, SNA, Function

class Model(GPSS2DEVS):
    def __init__(self):
        super().__init__("TelephoneExchange", 2)
        self.createStorage("LNKS", 10)
        self.createChain("WAIT")

        self.functions = {
            "FN$POISS": Function("RN1", Function.parse(
                "0.0,0.0/0.1,0.104/0.2,0.222/0.3,0.355/0.4,0.509/0.5,0.69/" \
                "0.6,0.915/0.7,1.2/0.75,1.38/0.8,1.6/0.84,1.83/0.88,2.12/" \
                "0.9,2.3/0.92,2.52/0.94,2.81/0.95,2.99/0.96,3.2/0.97,3.5/" \
                "0.98,3.9/0.99,4.6/0.995,5.3/0.998,6.2/0.999,7/0.9997,8", 24), True)
        }

        self.savevalues = {
            "XH$NOLINES": 50
        }

        self.variables = {
            "V$LINE": lambda: SNA("XH$NOLINES", self) * SNA("RN1", self) // 1000 + 1,
            "V$FREELN": lambda: SNA("XH$NOLINES", self) - 2 * SNA("S$LNKS", self) - SNA("CH$WAIT", self)
        }

        # The phone process
        self.GENERATE("L0", dist=lambda x: 12 * SNA("FN$POISS", self))
        self.TEST("L1", "V$FREELN", "G", "2")
        self.ASSIGN("ASN1", "PH$ORIG", lambda p: SNA("V$LINE", self))
        self.GATE("L3", "PH$ORIG", "ASN1", "R")
        self.ASSIGN("ASN2", "PH$DEST", lambda p: SNA("V$LINE", self))
        self.TEST("L5", "PH$ORIG", "NE", "PH$DEST")
        self.LOGIC("L6", "S", "PH$ORIG")
        self.ENTER("GETL", "LNKS")
        self.GATE("L8", "PH$DEST", "BUSY", "R")
        self.LOGIC("L9", "S", "PH$DEST")
        self.ADVANCE("L10", dist=lambda x:  120 * SNA("FN$POISS", self), prev="L9")
        self.LOGIC("L11", "R", "PH$ORIG")
        self.LOGIC("L12", "R", "PH$DEST")
        self.LEAVE("L13", "LNKS")
        self.TRANSFER_BOTH("L7", "GETL", "LNKS", self.GPSS2DEVS_5_L13.storage)
        self.TEST("CKCH", "CH$WAIT", "G", "0")
        self.GATE("L15", "1", "GETF", "R")
        self.UNLINK("L16", "WAIT", 1, "PH$DEST", "1", "GETF")
        self.TERMINATE("TERM")
        self.UNLINK("GETF", "WAIT", 1)

        self.TERMINATE("ABND")

        self.LINK("BLKD", "WAIT", "L7", "PH$ORIG")

        self.LOGIC("BUSY", "R", "PH$ORIG")
        self.LEAVE("L21", "LNKS")

        # The clock process
        self.GENERATE("L26", dist=lambda x: 60)
        self.TERMINATE("L27", "1")


        # CONNECTIONS

        self.connect("L0", "L1")
        self.connect("L1_Y", "ASN1")
        self.connect("L1_N", "ABND")
        self.connect("ASN1", "L3")
        self.connect("L3", "ASN2")
        self.connect("L3_B", "ASN1")
        self.connect("ASN2", "L5")
        self.connect("L5_Y", "L6")
        self.connect("L5_N", "ASN2")
        self.connect("L6", "L7")
        self.connect("L7", "BLKD")

        self.connect("GETL", "L8")
        self.connect("L8", "L9")
        self.connect("L8_B", "BUSY")
        self.connect("L9", "L10", True)
        self.connect("L10", "L11")

        self.connect("L11", "L12")
        self.connect("L12", "L13")
        self.connect("L13", "CKCH")
        self.connect("CKCH_Y", "L15")
        self.connect("CKCH_N", "TERM")
        self.connect("L15", "L16", True)
        self.connect("L15_B", "GETF", True)
        self.connect("L16", "TERM")
        self.connect("L16_B", "GETL")
        self.connect("L16_F", "GETF", True)
        self.connect("GETF", "TERM")
        self.connect("GETF_B", "GETL")
        self.connect("BUSY", "L21")
        self.connect("L21", "CKCH")

        self.connect("L26", "L27")
```

Figure 5.18: Python(P)DEVS code for *Example 5.4.2* (the telephone exchange).

# CHAPTER 6

# Wrapping Up

To conclude this thesis, we will discuss related work pertaining the topics we've discussed so far. Additionally, we will shortly revisit what we've done and draw some conclusions. Finally, we end with a description of possible expansions upon this research.

## 6.1 Related Work

The domain of Multi-Paradigm Modeling (MPM) [MV04] is continuously growing. It promotes the modeling of relevant aspects of a system at their most appropriate level(s) of abstraction, making use of the most relevant modeling language(s). Modeling language engineering techniques concerns themselves with the need for semantics in the evaluation of models.

DEVS can be seen as a common denominator for discrete-event modeling languages [Van00a]. It can therefore be used to describe the semantics of numerous modeling languages with some discrete-event abstraction. A Formalism Transformation Graph (FTG), a mapping between formalisms, is also discussed. Later on, the FTG is extended with a Process Model, resulting in the FTG+PM language [Luc+13] and allows the description of complex modeling workflows.

A mapping from Statecharts onto DEVS has been studied. [BV03] attempts to do a one-on-one transformation, similar to what we did for GPSS2DEVS; and [SV11] attempts to map the source model onto a single Atomic DEVS

for efficiency reasons. Moreover, [Dem14] describes how a CBD model can be mapped onto a single Atomic DEVS.

Despite its age, GPSS is still quite popular [Stå+11]. While most implementations don't seem to exist anymore, they paved the way for tools like GPSS World [Min10], aGPSS [Stå99] and GPSS/H [Cra97].

DEVS has a growing popularity with many frameworks, tools and libraries that implement the formalism (like Python(P)DEVS [Van14], adevs [Nut15]...). Additionally, the formalism can be easily transformed so that inexperienced programmers also understand what is going on under the hood [Mal+15].

## 6.2 Conclusions

In this thesis, we've started from the DEVS formalism and showed, while there are many tools that implement this formalism, most of them are lacking a formal description of basic models that can be used as a stepping stone in creating your own models. Furthermore, the tools that do provide such models only focus on the generation of items and the gathering of data.

Based on popular discrete-event tools that are used in an industrial context by major companies, we have created a building block library for DEVS, and more specifically, Python(P)DEVS. We focused on providing a wide enough basis for numerous M&S contexts, while, at the same time, not trying to oversaturate the library with functionality.

We distinguished seven generic parts in this BBL: *generators*, *queues*, *collectors*, *mathematics*, *input and output*, *transformations* and *routing*. Additionally, we provided some simulation tracers to help users analyze their models. With an example (that can also be found in the Python(P)DEVS documentation), we've shown the usefulness and applicability of the BBL.

Next, we based ourselves on the ideology that DEVS is a common denominator for discrete-event simulation formalisms and provided a valid transformation from a language in the process interaction world view, GPSS[1], onto DEVS, a formalism in the event scheduling world view. Despite the transformation between world view being non-trivial, the provided translation provides a contribution to the ongoing work in MPM.

This transformation followed the *three principles of GPSS2DEVS*: the *backwards propagation principle*, the *limitation principle* and the *availability principle*. They allowed us to create the transformation in a uniform manner, while at the same time remaining true to the process interaction world view. Additionally, they helped us constructing a translation that contains a traceability

[1] Technically, a subset thereof.

between the source and target model, opening the doors for complex features like debugging and testing.

Since we already had a BBL and GPSS2DEVS was also to be implemented in Python(P)DEVS, we were able to use some constructs of the BBL in this translation.

Finally, with some high-level and complex GPSS examples, we showed the validity of the translation w.r.t. existing GPSS tools like GPSS/H and GPSS World. The validity of this mapping allows us to benefit from the advantages of DEVS (e.g. modular models, scalable simulators, real-time execution...), while also exploiting the pros from GPSS (e.g. complex queueing systems, easily understandable system-flows, porwerful computation of complex models...).

## 6.3 Further Work

The work done in this thesis is by far complete and can easily be expanded in numerous ways. This section will discuss both the expansion on our BBL, as well as on the translation we have provided.

### 6.3.1 PythonDEVS-BBL

A building block library can be expanded indefinitely, because we can always find a functionality that is not yet supported. The two main goals of the library were its extensiveness and its completeness. Hence, we only focus on what is *common* and *useful*. That being said, there are some aspects of the library we can expand upon.

#### General Expansions

First things first, it has been made abundantly clear that PythonDEVS-BBL implements the CDEVS formalism. Special issues that occur due to superdense time may be solved by using the specialized SYNC block (*section 4.7*, *block 4.39*).

A clear expansion that can be done on the BBL is therefore the expansion to PDEVS. In the most ideal scenario, the library can be reused, with the only difference being the removal of the line

```
simulator.setClassicDEVS()
```

Unfortunately, at this point in time Python(P)DEVS has no way for models to know if they belong to a simulation according to CDEVS or PDEVS. Hence, this expansion will probably add this missing feature to the Python(P)DEVS simulation kernel.

Another expansion that can be done comes from an analysis of uses of the BBL. If we have enough different models, we may be able to deduce common structures of combinations of blocks from the BBL. To increase user-friendliness, these structures (as individual Coupled DEVS) may become part of a *snippet* part of the library, reducing possible overhead in creating models even further.

**Missing Building Blocks**

Both DEVS-Ruby and DEVSimPy provide building blocks that are able to write to specific file formats, yet we require two blocks for this purpose: the TRANSFORMER (*section 4.6*, *block 4.27*) and the FILE WRITER (*section 4.5*, *block 4.23*). Whereas some file formats, like CSV, are quite easy to create in this way, others, like XML, might become slightly more complex. Hence, a possible expansion upon the BBL is the creation of a specific FILETYPE WRITER (or set thereof) that is able to handle these requests. Additionally, while the full syslog protocol [Ger+09] is currently available in the BBL and many other possibilities were enabled via allowing the reading from and writing to files, tools like ExtendSim [Ima87] provide a full interaction with an external database, which is currently not cleanly supported in PythonDEVS-BBL.

Even though the SOUND block (*section 4.5*, *block 4.25*) is currently quite powerful for its purpose, a small expansion can be made in allowing users to choose a specific note or frequency to be played (similar as to what can be done in LEGO Mindstorms [LEG13]).

Finally, some sort of EQUATION SOLVER may provide even more strength to the BBL as a whole. Yet, this would require the need of either a CBD solver in DEVS (as in [Dem14]) or an additional library.

**Adding Domain-Specificity**

Whereas PythonDEVS-BBL has mainly focused on the generic kind of building blocks (i.e. the blocks that can be used in many different domains of modeling), we might also start considering some more domain-specific aspects.

For instance, when looking at ExtendSim [Ima87] FlexSim [Fle93] and Enterprise Dynamics [INC97], we notice that a lot of the building blocks they provide focus on industrial contexts. Conveyor belts, cranes, robots... they are all represented within their building block library. To allow PythonDEVS-BBL to be useful in those contexts, it might be interesting to add these features to the library in a domain-specific section.

Furthermore, *Agent-Based Modeling* (ABM) is an integral part of [Fle93], so building blocks for such a purpose can be as useful. To do this, we can follow the ABM specification that's proposed in [Ley20], which is already based on DEVS.

Additionally, [Dia+00] speaks of a *flow architecture* within their discrete formalism. Because our BBL cannot yet represent the flow of fluids, we might

be able to add this flow architecture and henceforth use a transformation between a continuous-time specification and DEVS, similar to [KJ01].

### 6.3.2 GPSS2DEVS

As far as the translation is concerned, the most obvious expansion is the implementation of the full GPSS semantics, instead of the subset that has been presented in this paper. Additionally, we could make use of some dependency analysis in order to optimize a model and the control messages that are being sent. Furthermore, the fact that our translation is traceable between the source and the target model, allows us to create a debugger for debugging the GPSS from the DEVS model, possibly making use of the DEVS debugger as is presented in [VVV18].

Alternatively, we could improve the efficiency of the target GPSS model by mapping the source model onto a single Atomic DEVS, similar to what was done in [SV11]. A drawback for this method is the disappearance of the traceability. Yet, in researching a way of mapping that still allows the traceability in the debugger that is presented in [VVV18], this drawback can be counteracted. This latter expansion on this research could also result in a brand new GPSS simulator that is able to parse and execute models for GPSS World, GPSS/H, aGPSS, GPSS V and HGPSS. The creation of such a tool may make it easier for new modelers to find their way to the GPSS formalism.

# Appendices

# APPENDIX A

# Distributions

This appendix goes into detail on the distributions that have been studied/implemented in PythonDEVS-BBL.

## A.1 Distribution Overview

The distributions that are available in PythonDEVS-BBL are selected from the distributions used in the available tools (see *section 3.1*) and the ones that were listed in [Law14]. This appendix provides an overview of the availability of these distribution w.r.t. the mentioned sources.

| Distribution | ExtendSim [Ima87] | SIMUL8 [SIM94] | FlexSim [Fle93] | Enterprise Dynamics [INC97] | LEGO Mindstorms [LEG13] | Simulation Modeling and Analysis [Law14] |
|---|---|---|---|---|---|---|
| Average (Normal with $\sigma = 0.25$) | | x | | | | |
| Bernoulli | | x | x | x | x | x |
| Beta | x | x | x | x | | x |
| Binomial | x | x | | | | x |
| Cauchy | x | | x | | | |
| Chi Squared | x | | | | | |
| Constant / Fixed | x | x | x | x | x | |
| Empirical | x | | x | | | x |
| Erlang | x | x | x | x | | x |
| Exponential | x | x | x | | | x |
| Extreme Value Type 1A | x | | x | | | |
| Extreme Value Type 1B | x | | x | | | |
| F | | | | | | |
| Gamma | x | x | x | x | | x |
| Geometric | x | x | | | | x |
| Hyper Exponential | x | | | | | |
| Hyper Geometric | x | | | | | |
| Inverse Gaussian | x | | x | | | |
| Inverse Weibull | x | | x | | | |
| Johnson Bounded | x | | x | | | |
| Johnson Unbounded | x | | x | | | |
| Laplace | x | | x | | | |
| Log-Laplace | | | x | | | |
| Log-Logistic | x | | x | | | |
| Log-Normal | x | x | x | x | | x |
| Logarithmic | x | | | | | |
| Logistic | x | | x | x | | |

| Distribution | ExtendSim [Ima87] | SIMUL8 [SIM94] | FlexSim [Fle93] | Enterprise Dynamics [INC97] | LEGO Mindstorms [LEG13] | Simulation Modeling and Analysis [Law14] |
|---|---|---|---|---|---|---|
| Negative Binomial | x | x | | | | x |
| Negative Exponential | | | | x | | |
| Normal | x | x | x | x | | x |
| Pareto | x | | x | | | |
| Pearson Type V | x | x | x | | | x |
| Pearson Type VI | x | x | x | | | x |
| Poisson | x | x | x | | | x |
| Powerfunction | x | | | | | |
| Random Walk | | | x | | | |
| Rayleigh | x | | | | | |
| Student's t | | | | | | |
| Triangular | x | x | x | x | | x |
| Uniform Integer | x | x | x | x | x | x |
| Uniform Real | x | x | x | x | | x |
| Weibull | x | x | x | x | | x |
| Zipf | | | | | | |

## A.2 Distributions in PythonDEVS-BBL

Based on the distributions from *appendix A.1*, this appendix provides a list of all the distributions that were made possible in the RNG building block (see *section 4.1.2*, *block 4.4*). For each distribution, the table lists a name, a description and how $F^{-1}$ was obtained w.r.t. the techniques listed in *section 4.1.2*.

| Name | Description | Technique or Formula |
|---|---|---|
| Bernoulli | The discrete Bernoulli distribution is a special case of the Binomial distribution where $n = 1$. | See Binomial. |
| Beta | The Beta distribution on interval $[0, 1]$ with shapes $a_1$ and $a_2$. | Any technique but closed form. |
| Binomial | Discrete Binomial distribution with $n$ experiments that have $p$ chance of success. | Inverse transform. |
| Cauchy | This distribution is also known as the Lorentz distribution or the Breit-Wigner distribution. It is often used as the canonical example of a pathological distribution. | $F^{-1}(y) = \gamma \cdot \tan(\pi \cdot (y - 0.5)) + x_0$ |
| $\chi^2$ | The $\chi^2$ distribution is a special case of the Gamma distribution with $a = k/2$ and $b = 1$. | See Gamma |
| Erlang | The Erlang distribution is a special case of the Gamma distribution where $b$ is an integer. | See Gamma |
| Exponential | Continuous exponential distribution with mean $\frac{1}{\lambda}$. The distribution can be used to predict the wait time until the first event.<br><br>We know the distribution is "memoryless", hence: $P(X > s + t \mid X > s) = P(X > t)$ | $F^{-1}(y) = -\frac{ln(y)}{\lambda}$ |

| Name | Description | Technique or Formula |
|---|---|---|
| Generalized Extreme Value Type A | This distribution is also known as the Gumbel distribution or the log-Weibull distribution. It is commonly used in hydrology. | $F^{-1}(y) = -\beta \ln(-\ln(y)) + \mu$ |
| Generalized Extreme Value Type B | This distribution is also known as the Fréchet distribution or the log-Weibull distribution. It is commonly used in hydrology. | $F^{-1}(y) = s(-\ln(y))^{-1/\alpha} + m$ |
| F | Fisher-Snedecor function with $d_1$ and $d_2$ its degrees of freedom. It uses the Beta distribution. | Closed form formula. |
| Gamma | Gamma distribution with shape $a$, rate $b$ and scale $\frac{1}{b}$. | Closed form formula w.r.t. preconditions |
| Geometric | Discrete Geometric distribution. | $F^{-1}(y) = \left\lfloor \frac{\ln(y)}{\ln(1-p)} \right\rfloor$ |
| Inverse Gauss | Also known as the Wald Distribution, it is commonly used to compute the properties of the Brownian Motion. | Inverse transformation with small enough step size. |
| Johnson Bounded | Johnson Bounded distribution with $a_1$ and $a_2$ the shape, $a$ the location and $b$ the scale. It uses the Standard Normal distribution. | Closed form formula, based on the Standard Normal distribution. |

| Name | Description | Technique or Formula |
|---|---|---|
| Johnson Unbounded | Johnson Unbounded distribution.<br>See also the Johnson Bounded distribution. | Closed form formula, based on the Standard Normal distribution. |
| Laplace | A distribution used in speech recognition and hydrology. | $F^{-1}(y) = \mu - b \cdot \text{sgn}(y - 0.5) \cdot \ln(1 - 2\|y - 0.5\|)$ |
| Log-Logistic | Fisk distribution that appears often in economical areas. | $F^{-1}(y) = b \cdot \left(\frac{y}{1-y}\right)^{1/a}$ |
| Log-Normal | Log-Normal distribution with mean $\mu$ and standard derivation $\sigma$.<br><br>The natural logarithm will be normally distributed w.r.t. $\mu$ and $\sigma$. | Closed form formula.<br>Can be obtained by taking the exponent of the corresponding Normal distribution. |
| Logistic | The distribution appears in logistic regression and feedforward neural networks. | $F^{-1}(y) = -s \cdot \ln(y^{-1} - 1) + \mu$ |
| Negative Binomial | Discrete Negative Binomial distribution with $p$ chance of success for each $s$ distributions.<br>This requires a set of random numbers.<br><br>Makes use of the Geometric distribution. | Closed form formula w.r.t. the Geometric distribution |

| Name | Description | Technique or Formula |
| --- | --- | --- |
| Normal | Normal distribution with mean $\mu$ and standard derivation $\sigma$. Use $\mu = 0$ and $\sigma = 1$ for the Standard Normal derivation. | Closed form formula based on the quantile function and the properties of the error function $erf$. |
| Pareto | The distribution is used to describe social, scientifical, geophysical, actuatial... phenomena. It originally was used to describe the distribution of wealth.<br><br>Colloquially, the distribution is refered to as the *80-20 rule*. | $F^{-1}(y) = x_m \cdot (1 - y)^{-1/\alpha}$ |
| Pearson Type V | Pearson Type V distribution with shape $a$ and scale $b$. It uses the Gamma distribution. | Inverse of the Gamma distribution with shape $a$ and scale $b$. |
| Pearson Type VI | Pearson Type VI distribution that combines two Gamma distributions. | Weighted value of two Gamma distributions. |
| Poisson | Discrete Poisson distribution with mean $\lambda$.<br><br>Expresses the probability that any given number of intervals occurs in a fixed interval of time. | Inverse transform. |

| Name | Description | Technique or Formula |
|---|---|---|
| Student's t | Student's t-distribution with $v$ degrees of freedom.<br>This distribution is often used for identifying unknowns in any distribution.<br>Within the context of RNG, they appear in multi-dimensional applications of copula-dependency. | Inverse transformation with small enough step size. |
| Triangular | Triangular distribution with minumum $a$, mode $b$ and maximum $c$.<br>This distribution is typically used as a surjective description of a population for which there is limited sample data. | if $y < (c-a)/(b-a)$,<br>$F^{-1}(y) = a + \sqrt{y(b-a)(c-a)}$<br>otherwise,<br>$c - \sqrt{(1-y)(c-a)(c-b)}$ |
| Uniform Integer | Rounded (discrete) uniform distribution over the range $[a, b]$. | $F^{-1}(y) = a + \lfloor (b-a+1) \cdot y \rfloor$ |
| Uniform Real | Continuous uniform distribution over the range $(a, b)$. | $F^{-1}(y) = a + ((b-a) \cdot y)$ |
| Weibull | The Weibull distribution with scale $a$, shape $b$ and location $v$.<br>Commonly used to predict the lifetime of technical equipment.<br>Note that $x - v$ must be strictly positive. | $F^{-1}(y) = v + a \cdot (-\ln(y))^{1/b}$ |
| Zipf | Discrete distribution w.r.t. Zipf's Law.<br>Makes use of harmonic numbers. | Inverse transformation. |

# APPENDIX B

## Building Block Port Information

Throughout this thesis, many building blocks were mentioned. This appendix provides additional information on the input and output sets for these blocks. For each block, a summary is given for the names for the ports, their type and/or range and their meaning. The order in which they are listed corresponds to the order in which they occur in *chapters 4* and *5*.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *value* | Any | A new value to be outputted. |
| | *dt* | $\mathbb{R}^{+}_{+\infty} \setminus \{0\}$ | A new time delay between the generated messages. |
| Outputs | *output* | Any | The generated values. |

Table B.1: Ports for the CONSTANT GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *dt* | $\mathbb{R}^{+}_{+\infty} \setminus \{0\}$ | A new time delay between the generated messages. |
| Outputs | *output* | Any | The generated values. |

Table B.2: Ports for the FUNCTION GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Outputs | *output* | Any | The generated values. |

Table B.3: Ports for the TABLE GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Outputs | *output* | Any | The value that's outputted at the predefined time. |

Table B.4: Ports for the SINGLE FIRE building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | $\mathbb{N}$ | The amount of items to output in bulk. |
| | *dt* | $\mathbb{R}^{+}_{+\infty} \setminus \{0\}$ | A new time delay between the generated messages. |
| Outputs | *output* | Any | The generated values. |

Table B.5: Ports for the BULK GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *dt* | $\mathbb{R}^{+}_{+\infty} \setminus \{0\}$ | A new time delay between the generated messages. |
| Outputs | *output* | $\mathbb{R}_{-\infty,+\infty}$ | The generated values. |

Table B.6: Ports for the RANDOM NUMBER GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *halt* | Boolean | Whether to halt/continue the generator. |
| Outputs | *output* | Any | The generated values. |

Table B.7: Ports for the RANDOM DELAY GENERATOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *enqueue* | Any | The item to enqueue. |
| | *dr* | $\mathbb{R}^+_{+\infty} \setminus \{0\}$ | The maximal delay an item may spend in the queue. |
| | *requestdequeue* | $\mathbb{N}$ or Any | Indicates that items must dequeue. When the block is constructed such that the arriving value indicates the amount of dequeues that must happen, it expects a natural number. Otherwise, any arriving value is accepted and a single item will be dequeued. |
| | *requestrenege* | $\mathbb{N}$ or Any | Indicates that the last item in the queue must depart on the *renege* port. When the block is constructed such that the arriving value indicates the amount of reneges that must happen, it expects a natural number. Otherwise, any arriving value is accepted and a single item will be reneged. |
| Outputs | *dequeue* | Any | The items that are dequeued. |
| | *renege* | Any | The items that are reneged. |
| | *overflow* | Any | The items that were enqueued if the block has already reached a maximal capacity. |

Table B.8: Ports for the Simple Queue building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *dt* | $\mathbb{R}^+_{+\infty} \setminus \{0\}$ | The time delay for periodic output. |
| | *enter* | Any | The item to enter the queue. |
| | *leave* | Any | The item to leave the queue. |
| Outputs | *size* | $\mathbb{N}$ | The size of the queue. |

Table B.9: Ports for the Queue Tracker building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *enqueue* | Any | The item to enqueue. |
| | *dd* | $\mathbb{R}^+_{+\infty} \setminus \{0\}$ | The delay between two dequeues. |
| | *dr* | $\mathbb{R}^+_{+\infty} \setminus \{0\}$ | The maximal delay an item may spend in the queue. |
| | *requestdequeue* | $\mathbb{N}$ or Any | Indicates that items must dequeue. When the block is constructed such that the arriving value indicates the amount of dequeues that must happen, it expects a natural number. Otherwise, any arriving value is accepted and a single item will be dequeued. |
| | *requestrenege* | $\mathbb{N}$ or Any | Indicates that the last item in the queue must depart on the *renege* port. When the block is constructed such that the arriving value indicates the amount of reneges that must happen, it expects a natural number. Otherwise, any arriving value is accepted and a single item will be reneged. |
| Outputs | *dequeue* | Any | The items that are dequeued. |
| | *renege* | Any | The items that are reneged. |
| | *overflow* | Any | The items that were enqueued if the block has already reached a maximal capacity. |
| | *count* | $\mathbb{N}$ | The size of the queue. |

Table B.10: Ports for the QUEUE building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | The item that needs to be remembered. |
| Outputs | *current* | Any | The current item. |
| | *previous* | Any | The previous item, when in *individual* mode, or all items from the same time instance when in *collection* mode. |

Table B.11: Ports for the RETAIN building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | The item to schedule, must arrive at the same time as the item on the *delay* port. |
| | *delay* | $\mathbb{R}^{+}_{+\infty}$ | The delay for the item that arrived on the *input* port. |
| | *pause* | List of (`item`, Boolean) | Indicates that certain items in the block must be paused, while others may be continued. |
| Outputs | *output* | Any | The items that are delayed for a specific delay. |

Table B.12: Ports for the ADVANCE building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | The items to obtain. |
| | *clear* | Any | Clears the contents of the block. |

Table B.13: Ports for the TABLE COLLECTOR building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | $\mathbb{R}$ | The values to measure. |
| | *clear* | Any | Clears the statistics of the block. |

Table B.14: Ports for the COLLECTOR and ESTIMATE COLLECTOR building blocks.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | The messages to count. |
| | *clear* | Any | Clears the contents of the block. |
| Outputs | *output* | Any | The counted item. |
| | *count* | $\mathbb{N}$ | The amount of messages that have passed through. |

Table B.15: Ports for the COUNTER building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input-1* | $\mathbb{R}_{+\infty}$ | The terms to add together. |
| | *input-2* | | |
| | ... | | |
| | *input-n* | | |
| Outputs | *result* | $\mathbb{R}_{+\infty}$ | The sum of all inputs. |

Table B.16: Ports for the ADDER building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input-1* | $\mathbb{R}_{+\infty}$ | The factors to multiply with each other. |
| | *input-2* | | |
| | ... | | |
| | *input-n* | | |
| Outputs | *result* | $\mathbb{R}_{+\infty}$ | The product of all inputs. |

Table B.17: Ports for the MULTIPLIER building block.

|  | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input-1* | $\mathbb{R}_{+\infty}$ | All parameters of the equation. The names for the ports are defined by the argument names for the function that's associated with the block. |
|  | *input-2* |  |  |
|  | ... |  |  |
|  | *input-n* |  |  |
| Outputs | *result* | $\mathbb{R}_{+\infty}$ | The result of the equation. |

Table B.18: Ports for the EQUATION building block.

|  | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | $\mathbb{R}_{+\infty}$ | The value to differentiate. |
| Outputs | *result* | $\mathbb{R}_{+\infty}$ | The differentiated value. |

Table B.19: Ports for the DIFFERENTIATOR building block.

|  | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | $\mathbb{R}_{+\infty}$ | The value to integrate. |
| Outputs | *result* | $\mathbb{R}_{+\infty}$ | The integrated value. |

Table B.20: Ports for the INTEGRATOR building block.

|  | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | An item that triggers the generation of a new random number. |
| Outputs | *random* | $\mathbb{R}_{-\infty,+\infty}$ | A random value, i.i.d. the provided distribution. |
|  | *value* | Any | The item that was inputted. |

Table B.21: Ports for the RANDOM building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | String | The message to be logged. |

Table B.22: Ports for the Logger building block and all its derivatives.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | String or Bytes | The message to be written to a file, may be binary. |

Table B.23: Ports for the File Writer building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *filename* | String | The name of the file to read, either as an absolute path, or relative to the execution directory. |
| Outputs | *contents* | String or Bytes | The contents of the file, possibly binary. |

Table B.24: Ports for the File Reader building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *dt* | $\mathbb{R}^{+}_{+\infty} \setminus \{0\}$ | A polling delay. |
| Outputs | *listen* | Any | The return value of the associated function. |

Table B.25: Ports for the Listener building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | Item that is checked against a condition, before the sound is played. |

Table B.26: Ports for the Sound building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | The item to transform. |
| Outputs | *transformed* | Any | The transformed value. |
| | *original* | Any | The original value that was inputted in the block. |

Table B.27: Ports for the TRANSFORMER and LOOKUP TABLE building blocks.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | A new item to add to the package. |
| | *release* | Any | Requests the created package to be released. |
| Output | *output* | `Package` | A packaged object. |

Table B.28: Ports for the PACK building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | `Package` | A packaged object that needs to be extracted. |
| Outputs | *output* | List | The contents of the package, as a list. |

Table B.29: Ports for the UNPACK building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | Item that has finished simulating. |

Table B.30: Ports for the FINISH and HALT building blocks.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *select* | ℕ | The index of the input to choose. |
| | *input-1* | Any | A list of possible inputs to route over the *output* port. |
| | *input-2* | | |
| | ... | | |
| | *input-n* | | |
| Outputs | *output* | Any | The item that was inputted on the selected input. |

Table B.31: Ports for the CHOOSE INPUT building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *select* | ℕ | The index of the input to choose. |
| | *input* | Any | The input to be routed conditionally over the selected output. |
| Outputs | *output-1* | Any | A list of possible outputs to route the input over. |
| | *output-2* | | |
| | ... | | |
| | *output-n* | | |

Table B.32: Ports for the CHOOSE OUTPUT building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *claim* | Any | Marks the currently selected path as "busy". |
| | *input-1* | Any | Indicates that the corresponding path has become free again. |
| | *input-2* | | |
| | ... | | |
| | *input-n* | | |
| Outputs | *output* | ℕ | The index of the path to select. Is to be inputted in the *select* port of the CHOOSE OUTPUT block. |
| | *free* | Boolean | Outputs `true` when there is at least one free path. |

Table B.33: Ports for the PICK building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | Items that require access to the resource. |
| | *leave* | Any | The items that release access to the resource. |
| Outputs | *blocked* | Any | The item that entered on the *input* port when no access to the resource is possible. |
| | *guarded* | Any | The items that were granted access to the resource. |
| | *unguarded* | Any | The items that released access to the resource, i.e. the items that entered on the *leave* input. |

Table B.34: Ports for the GUARD building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | Any | Items want to gain access to a critical section. |
| | *block* | Boolean | When `true`, the gate will be closed. |
| Outputs | *output* | Any | All items that arrived on *input* when the gate is open. |
| | *blocked* | Any | All items that arrived on *input* when the gate is closed. |

Table B.35: Ports for the GATE building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *start* | Any | Starts the timer for the item. |
| | *dt* | $\mathbb{R}^{+}_{+\infty}$ | A new delay to hold an item for. |
| Outputs | *blocked* | Any | All items that arrived when the block was busy. |
| | *finished* | Any | The item that was delayed. |

Table B.36: Ports for the TIMER building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *start* | Any | Starts the timer for the item. |
| Outputs | *blocked* | Any | All items that arrived when the block was busy. |
| | *finished* | Any | The item that was delayed. |

Table B.37: Ports for the DELAYER building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input-1* | Any | Items that must be delayed until all others are received. |
| | *input-2* | | |
| | ... | | |
| | *input-n* | | |
| Outputs | *output-1* | Any | All obtained items from all inputs. |
| | *output-2* | | |
| | ... | | |
| | *output-n* | | |

Table B.38: Ports for the SYNC building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *create* | `Transaction` | The newly generated transaction. |
| | *terminate* | (`Transaction`, ℕ) | A transaction that needs to be terminated and decrease the termination counter with a certain amount. |
| | *delay* | `Transaction` | A transaction that enters an ADVANCE or a LINK block. |
| | *moved* | `Transaction` or (`Transaction`, Boolean) or List of (`Transaction`, Boolean) | A transaction that has moved, possibly with its blocked status or a group of transactions with their blocked statuses. |
| | *unblock* | `Transaction` | A transaction that is not blocked anymore. |
| | *facilities* | (String, `Transaction`, Boolean) | Updated value of a facility. |
| Outputs | *notify* | `Transaction` | The transaction that is allowed to move. |
| | *pause* | List of (`item`, Boolean) | All transactions that need to be paused in the corresponding ADVANCE block. |

Table B.39: Ports for the CONTROLLER building block.

| | Name | Type / Range | Description |
|---|---|---|---|
| Inputs | *input* | `Transaction` | A transaction that needs to be passed on. |
| | *select* | Boolean | When `true` at the time a transaction arrives on the *input* port, the transaction needs to be blocked. |
| | *release* | `Transaction` | The transaction that is allowed to move. |
| Outputs | *output* | `Transaction` | The transaction that is allowed to move, if it was located in the block. |
| | *fallback* | `Transaction` | The transaction that is allowed to move after it was blocked. |
| | *contents* | List of (`Transaction`, Boolean) | The contents of the block, i.e. all transactions that are waiting at this point, together with their blocked status. |

Table B.40: Ports for the HOLD building block.

# APPENDIX C

# GPSS Blocks

Different tools implement different functionalities and subsets of the GPSS building block set. This appendix provides an overview of these subsets, based on [CC09]. An "x" indicates that the block is available in the given tool and an "(x)" means that the block is only partially available.

| | GPSS World [Min10] | GPSS/H [Cra97] | WebGPSS aGPSS [Stå99] | JGPSS [CC09] | GPSS2DEVS |
|---|---|---|---|---|---|
| ADOPT | x | | | | |
| ADVANCE | x | x | x | x | x |
| ALTER | x | x | | | x |
| ATNWAIT | | x | | | |
| ARRIVE | | | x | | |
| ASSEMBLE | x | x | x | x | |
| ASSIGN | x | x | | | x |
| BRANCH | | | | | |
| BUFFER | x | x | | x | |
| BCLOSE | | x | | | |
| BFILEDEF | | x | | | |

| | GPSS World [Min10] | GPSS/H [Cra97] | WebGPSS aGPSS [Stå99] | JGPSS [CC09] | GPSS2DEVS |
|---|---|---|---|---|---|
| BGETLIST | | x | | | |
| BGETSTRING | | x | | | |
| BLET | | x | | | |
| BPUTPIC | | x | | | |
| BPUTSTRING | | x | | | |
| BCLEAR | | x | | | |
| BCALL | | x | | | |
| BRESET | | x | | | |
| CHANGE | | x | | | |
| CLOSE | x | | | | |
| COMPARE | | | | | |
| COUNT | x | x | | | |
| DEPART | x | x | x | x | x |
| DISPLACE | x | | | | |
| ENTER | x | x | x | x | x |
| EXAMINE | x | x | | | |
| EXECUTE | x | x | | | |
| FAVAIL | x | x | | x | |
| FUNAVAIL | x | x | | x | |
| GATE | x | x | | x | x |
| GATHER | x | x | | x | |
| GENERATE | x | x | x | x | x |
| GOTO | | | x | | |
| HELP | | x | x | | |
| IF | | | x | | |
| INDEX | x | x | | | |
| INTEGRATION | x | | | | |
| JOIN | x | x | | | |
| LEAVE | x | x | x | x | x |
| LET | | | x | | |
| LINK | x | x | | | x |
| LOGIC | x | x | | x | x |

| | GPSS World [Min10] | GPSS/H [Cra97] | WebGPSS aGPSS [Stå99] | JGPSS [CC09] | GPSS2DEVS |
|---|---|---|---|---|---|
| LOOP | x | x | | x | |
| MARK | x | x | | | x |
| MATCH | x | x | | x | |
| MSAVEVALUE | x | x | | x | |
| OPEN | x | | | | |
| PLUS | x | | | | |
| PREEMPT | x | x | | | x |
| PRINT | | x | x | | |
| PRIORITY | x | x | | x | |
| QUEUE | x | x | | x | x |
| READ | x | | | | |
| RELEASE | x | x | x | x | x |
| REMOVE | x | x | | | |
| RESCHEDULE | | x | | | |
| RETURN | x | x | | | x |
| SAVAIL | x | x | | x | |
| SAVEVALUE | x | x | | x | |
| SCAN | x | x | | | |
| SEEK | x | | | | |
| SEIZE | x | x | x | x | x |
| SELECT | x | x | | | |
| SPLIT | x | x | x | x | |
| SUNAVAIL | x | x | | x | |
| TABULATE | x | x | x | | x |
| TERMINATE | x | x | x | x | x |
| TEST | x | x | | x | x |
| TRACE | x | x | | | |
| TRANSFER | x | x | | (x) | (x) |
| UNLINK | x | x | | | x |
| UNTRACE | x | x | | | |
| WAITIF | | | x | | |
| WRITE | x | | | | |

# Bibliography


[ATT91] AT&T Labs Research. *Graphviz*. https://www.graphviz.org/. 1991. Used in 2020.

[Aut96] Autodesk. *3ds Max*. https://www.autodesk.com/products/3ds-max/. 1996. Accessed 08-06-2020.

[Aut98] Autodesk. *Maya 3D*. https://www.autodesk.com/products/maya/. 1998. Accessed 08-06-2020.

[Bar+11] A. Barišić et al. "Quality in Use of Domain-specific Languages: A Case Study". In: *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU '11. ACM, 2011, pages 65–72.

[Bar95] F. J. Barros. "Dynamic Structure Discrete Event System Specification: a New Formalism for Dynamic Structure Modeling and Simulation". In: *Proceedings of the Winter Simulation Conference*. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc., 1995, pages 781–785.

[Bar97] F. J. Barros. "Modeling Formalisms for Dynamic Structure Systems". In: *ACM Transactions on Modeling and Computer Simulation* 7.4 (Oct. 1997), pages 501–515.

[Bar98] F. J. Barros. "Abstract Simulators for the DSDE Formalism". In: *Proceedings of the 1998 Winter simulation Conference*. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc., 1998, pages 407–412.

[Bel11] F. Bellard. *FFmpeg File Formats*. http://www.ffmpeg.org/general.html#File-Formats. 2011. Online; accessed 01-06-2020.

[Ble98] Blender Foundation. *Blender*. https://www.blender.org/. 1998. Accessed 08-06-2020.

[BV03] S. Borland and H. Vangheluwe. "Transforming statecharts to DEVS". In: *Summer Computer Simulation Conference (Student Workshop)*. 2003, S154–S159.

[Cap+11] L. Capocchi et al. "DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems". In: *IEEE 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. June 2011, pages 170–175.

[Cap19] L. Capocchi. *DEVSimPy*. https://github.com/capocchi/DEVSimPy. 2019. Online; accessed 31-05-2020.

[CC09] P. F. i Casas and J. Casanovas. "JGPSS, an open source GPSS framework to teach simulation". In: *Proceedings of the 2009 Winter Simulation Conference*. 2009.

[Cla92] F. Claeys. "HGPSS: Object-geörienteerde "process-interaction" simulatie". Master's thesis. University of Ghent, June 1, 1992.

[Cra97] R. C. Crain. "Simulation using GPSS/H". In: *Proceedings of the 29th conference on Winter simulation - WSC 1997*. ACM Press, 1997.

[CV10] B. Chen and H. Vangheluwe. "Symbolic Flattening of DEVS models". In: *Proceedings of the 2010 Summer Simulation Multiconference*. 2010, pages 209–218.

[CZ94] A. C. H. Chow and B. P. Zeigler. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". In: *Proceedings of Winter Simulation Conference*. IEEE. 1994, pages 716–722.

[Dem14] N. Demarbaix. *Causal Block Diagram: compiler to LaTeX and DEVS*. Research report. University of Antwerp, 2014.

[Dev86] L. Devroye. "General Principles in Random Variate Generation". In: *Non-Uniform Random Variate Generation*. Springer Science + Business Media New York, 1986, pages 27–82.

[Dia+00] B. Diamond et al. *Extend v5 User's Guide*. Edited by Imagine That, Inc. 2000.

[Dro+03] M. Droettboom et al. *Matplotlib*. https://matplotlib.org/. 2003. Online; accessed 09-06-2020.

[Epi98] Epic Games. *Unreal Engine*. https://www.unrealengine.com/en-US/. 1998. Accessed 08-06-2020.

[Eur+66] European Computer Manufacturers Association et al. "Standard ECMA-4: Flow charts". In: *European Computer Manufacturers Association* (1966).

[Fau19] A. C. Faul. "Sampling". In: *A Concise Introduction to Machine Learning*. CRC Press, 2019, pages 63–66.

[FB04] J.-B. Filippi and P. Bisgambiglia. "JDEVS: an implementation of a DEVS based formal framework for environmental modelling". In: *Environmental Modelling & Software* (2004), pages 261–274.

[Fle93] FlexSim Software Products, Inc. *FlexSim*. https://www.flexsim.com/. 1993. Used in 2020.

[FP87] J. E. Freund and B. M. Perles. "A New Look at Quartiles of Ungrouped Data". In: *The American Statistician* 41.3 (1987), pages 200–203.

[Fra+14] R. Franceschini et al. "DEVS-Ruby: A Domain Specific Language for DEVS Modeling and Simulation (WIP)". In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS*. Tampa, FL, USA, 2014, pages 103–108.

[Fra+18] R. Franceschini et al. "An Overview of the Quartz Modelling and Simulation Framework". In: *Proceedings of 8th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Volume 19. 3. Science and Technology Publications, 2018.

[FS07] D. Feldman and Y. Shavitt. "An optimal median calculation algorithm for estimating Internet link delays from active measurements". In: *Workshop on End-to-End Monitoring Techniques and Services*. IEEE. 2007, pages 1–7.

[GBK16] R. Goldstein, S. Breslav, and A. Khan. "DesignDEVS: Reinforcing Theoretical Principles in a Practical and Lightweight Simulation Environment". In: *Proceedings of the 2016 Spring Simulation Multiconference*. Pasadena, CA, USA, 2016, pages 1–8.

[GDV16] C. Gomes, J. Denil, and H. Vangheluwe. *Causal-block diagrams*. Research report. University of Antwerp, 2016.

[Ger+09] R. Gerhards et al. *The syslog protocol*. Technical report. Network Working Group, 2009.

[GLV07] H. Giese, T. Levendovszky, and H. Vangheluwe. "Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools". In: *Models in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 252–262.

[Gor75] G. Gordon. *The Application of GPSS V to Discrete System Simulation*. Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[Gor78a] G. Gordon. *System Simulation*. 2nd edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

[Gor78b] G. Gordon. "The Development of the General Purpose Simulation System (GPSS)". In: *History of programming languages* 13.8 (Aug. 1978), pages 183–198.

[Gou69] R. L. Gould. "GPSS/360 - An improved general purpose simulator". In: *IBM Systems Journal* 8.1 (1969), pages 16–27.

[Gue+10] E. Guerra et al. "A Visual Specification Language for Model-to-Model Transformations". In: *IEEE Symposium on Visual Languages and Human-Centric Computing*. 2010.

[Gue+13] E. Guerra et al. "Automated verification of model transformations based on visual contracts". In: *Automated Software Engineering* 20.1 (2013), pages 5–46.

[Ham15] J. Hamilton. *SimpleAudio*. https://simpleaudio.readthedocs.io/. 2015. Online; accessed 01-06-2020.

[Har87] D. Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pages 231–274.

[Hek16] M. Hekimoğlu. *IE 303, Discrete-Event Simulation, Lecture 5: Probability Review*. http://www2.isikun.edu.tr/mustafahekimoglu/simulation/Lecture5-ProbabilityReview.pdf. 2016. Online; accessed 07-12-2019.

[Hol97] G. J. Holzmann. "The Model Checker SPIN". In: *Transactions on Software Engineering* 23.5 (1997), pages 279–295.

[HR04] D. Harel and B. Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10 (2004), pages 64–72.

[Hwa07a] M. H. Hwang. *DEVS#: C# open source library of DEVS formalism*. http://xsy-csharp.sourceforge.net/DEVSsharp/. May 2007. Online; accessed 31-05-2020.

[Hwa07b] M. H. Hwang. *DEVS++: C++ open source library of DEVS formalism*. http://odevspp.sourceforge.net/. 2007. Online; accessed 31-05-2020.

[Hwa07c] M. H. Hwang. *Modeling and Simulation using DEVS#*. 2007.

[Ima87] Imagine That, Inc. *ExtendSim*. https://www.extendsim.com/. 1987. Used in 2020.

[INC97] INCONTROL Simulation Solutions. *Enterprise Dynamics*. https://www.incontrolsim.com/. 1997. Used in 2020.

[JC85] R. Jain and I. Chlamtac. “The P2 algorithm for dynamic calculation of quantiles and histograms without storing observations”. In: *Communications of the ACM* 28.10 (1985), pages 1076–1085.

[Kar+14] A. T. Karl et al. “Using RngStreams for parallel random number generation in C++ and R”. In: *Computational Statistics* 29.5 (2014), pages 1301–1320.

[Ken53] D. G. Kendall. “Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain”. In: *The Annals of Mathematical Statistics* 24.3 (1953), pages 338–354.

[KJ01] E. Kofman and S. Junco. “Quantized-state systems: a DEVS Approach for continuous system simulation”. In: *Transactions of The Society for Modeling and Simulation International* 18.3 (2001), pages 123–132.

[KK00] J.-H. Kim and T. G. Kim. “Framework for modeling/simulation of mobile agent systems”. In: *Proceedings of 2000 Conference on AI, Simulation and Planning in High Autonomy Systems*. Citeseer. 2000, pages 53–59.

[Kle07] A. Kleppe. “A language description is more than a metamodel”. In: *Fourth International Workshop on Software Language Engineering*. 2007.

[Kna10] T. Knaresboro. *Popular Mechanics: How to Choose the Fastest Line*. https://www.popularmechanics.com/science/health/a6164/how-to-choose-the-fastest-line/. 2010. Online; accessed 11-12-2019.

[KSE09] S. Kim, H. S. Sarjoughian, and V. Elamvazhuthi. “DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring”. In: *Proceedings of the 2009 Spring Simulation Multiconference*. San Diego, CA, USA, 2009, pages 1–7.

[Küh+10] T. Kühne et al. “Explicit Transformation Modeling.” In: *MoDELS Workshops*. Volume 6002. Lecture Notes in Computer Science. Springer, Apr. 13, 2010, pages 240–255.

[Küh06] T. Kühne. “Matters of (Meta-)Modeling”. In: *Software and System Modeling* 5 (2006), pages 369–385.

[Kwo+96] Y. W. Kwon et al. "Fuzzy-DEVS Formalism: Concepts, Realization and Application". In: *Proceedings of AI, Simulation and Planning in High Autonomy Systems*. San Diego, CA, USA, 1996, pages 227–234.

[Lan06] E. Langford. "Quartiles in Elementary Statistics". In: *Journal of Statistics* 14.3 (2006).

[Lar87] R. C. Larson. "OR Forum – Perspectives on Queues: Social Justice and the Psychology of Queueing". In: *Operations Research* 35.6 (1987), pages 895–905.

[Law14] A. M. Law. "Generating Random Variates". In: *Simulation Modeling and Analysis, Fifth Edition*. McGraw-Hill Education, 2014, pages 426–487.

[LEc+02] P. L'Ecuyer et al. "An object-oriented random-number package with many long streams and substreams". In: *Operations research* 50.6 (2002), pages 1073–1075.

[LEc01] P. L'Ecuyer. "Random Numbers". In: *Int. Encyc. Social and Behavioural Sciences* (2001).

[LEc12] P. L'Ecuyer. "Random Number Generation". In: *Handbook of computational statistics*. Springer Berlin Heidelberg, 2012, pages 35–71.

[LEG13] LEGO A/S. *LEGO Mindstorms EV3*. https://mindstorms.lego.com/. 2013. Used in 2020.

[Ley20] T. Leys. "Towards an Agent-Based Modeling Platform with Precise Semantics". Master's thesis. University of Antwerp, Jan. 1, 2020.

[Lia11] P.-Y. Liao. "A queuing model with balking and reneging rate". In: *International Journal of Services and Operations Management* 10.1 (Aug. 22, 2011).

[LS03] P. L'Ecuyer and R. Simard. *RngStreams: An object-oriented random-number package in C with many long streams and substreams*. http://statmath.wu.ac.at/software/RngStreams/. 2003. Online; accessed 31-05-2020.

[LS07] P. L'Ecuyer and R. Simard. "TestU01: A C library for empirical testing of random number generators". In: *ACM Transactions on Mathematical Software (TOMS)* 33.4 (2007), pages 1–40.

[Luc+13] L. Lucio et al. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In: *SDL: Model-Driven Dependability Engineering*. Volume 7916. LNCS. 2013, pages 182–202.

[LV02a] J. de Lara and H. Vangheluwe. "AToM$^3$: A Tool for Multi-formalism and Meta-modelling". In: *Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 15, 2002, pages 174–188.

[LV02b] J. de Lara and H. Vangheluwe. "Using Meta-Modelling and Graph Grammars to process GPSS models". In: 16$^{th}$ *European Simulation Multi-conference (ESM)*. SCS, June 2002, pages 100–107.

[Mal+15] M. Maleki et al. "Designing DEVS visual interfaces for end-user programmers". In: *Simulation* 91.8 (2015), pages 715–734.

[Max90] Maxon. *Cinema 4D*. https://www.maxon.net/en-us/products/cinema-4d/overview/. 1990. Accessed 08-06-2020.

[Mel16] C. Mele. *How to Pick the Fastest Line at the Supermarket*. https://www.nytimes.com/2016/09/08/business/how-to-pick-the-fastest-line-at-the-supermarket.html. 2016. Online; accessed 11-12-2019.

[Mic87] Microsoft. *Microsoft Excel*. https://www.microsoft.com/nl-be/microsoft-365/excel. 1987. Accessed 09-06-2020.

[Min10] Minuteman Software. *GPSS World*. http://www.minutemansoftware.com/. 2010. Used in 2020.

[Min72] Minitab, LLC. *Minitab*. https://minitab.com/. 1972. Accessed 09-06-2020.

[MM03] D. S. Moore and G. P. McCabe. *Introduction to the Practice of Statistics*. 4th edition. 2003.

[MN05] A. Muzy and J. J. Nutaro. "Algorithms for Efficient Implementations of the DEVS & DSDEVS Abstract Simulators". In: *1st Open International Conference on Modeling and Simulation*. 2005, pages 273–279.

[Moo09] D. Moody. "The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering". In: *IEEE Transactions on Software Engineering* 35.6 (2009), pages 756–779.

[Mur89] T. Murata. "Petri nets: Properties, Analysis and applications". In: *Proceedings of the IEEE* 77.4 (1989), pages 541–580.

[MV04] P. J. Mosterman and H. Vangheluwe. "Computer Automated Multi-Paradigm Modeling: An Introduction". In: *Simulation* 80.9 (Sept. 1, 2004), pages 433–450.

[MV14] B. Meyers and H. Vangheluwe. “A Multi-Paradigm Modelling Approach for the Engineering of Modelling Languages”. In: *Proceedings of the Doctoral Symposium of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*. CEUR Workshop Proceedings. 2014, pages 1–8.

[Nut10] J. J. Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. 1st edition. Hoboken, NJ, USA: Wiley, 2010.

[Nut15] J. J. Nutaro. *adevs*. http://www.ornl.gov/~1qn/adevs/. 2015. Online; accessed 10-05-2020.

[Oli+95] T. Oliphant et al. *NumPy*. https://numpy.org/. 1995. Online; accessed 09-06-2020.

[ON04] C. M. Overstreet and R. E. Nance. “Characterizations and Relationships of World Views”. In: *Proceedings of the 36th Conference on Winter Simulation*. WSC ’04. Washington, D.C.: Winter Simulation Conference, 2004, pages 279–287.

[Pet77] J. L. Peterson. “Petri Nets”. In: *ACM Computing Surveys (CSUR)* 9.3 (1977), pages 223–252.

[Pri19] J. Prisco. *Popular Mechanics: How to Choose the Fastest Line*. https://edition.cnn.com/style/article/design-of-waiting-lines/index.html. Feb. 15, 2019. Online; accessed 08-06-2020.

[Pyt] Python Software Foundation. *Python Language Reference, version 3*. https://www.python.org/. Online; accessed 13-02-2020.

[R F97] R Foundation. *R*. https://www.r-project.org/. 1997. Accessed 09-06-2020.

[Res+09] M. Resnick et al. “Scratch: programming for all”. In: *Communications of the ACM* 52.11 (2009), pages 60–67.

[Ris+17] J. L. Risco-Martín et al. “Reconsidering the Performance of DEVS Modeling and Simulation Environment Using the DEVStone Benchmark”. In: *Simulation* (2017).

[Ris10] M. Risoldi. “A methodology for the development of complex domain-specific languages”. PhD thesis. University of Geneva, June 7, 2010.

[Rob11] J. Robert. *Pydub*. http://pydub.com/. 2011. Online; accessed 01-06-2020.

[Sch74] T. J. Schriber. *Simulation Using GPSS*. Wiley, 1974.

[She08] M. Shermer. “Patternicity: Finding Meaningful Patterns in Meaningless Noise”. In: *Scientific American* (2008).

[SIM94] SIMUL8 Corporation. *SIMUL8*. https://www.simul8.com/. 1994. Used in 2020.

[SK03] S. Sendall and W. Kozaczynski. “Model Transformation: The Heart and Soul of Model-Driven Software Development”. In: *IEEE Software* 20.5 (2003), pages 42–45.

[SS15] H. S. Sarjoughian and S. Sundaramoorthi. “Superdense Time Trajectories for DEVS Simulation Models”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. DEVS ’15. Alexandria, Virginia: Society for Computer Simulation International, 2015, pages 249–256.

[Stå+11] I. Ståhl et al. “GPSS 50 years old, but still young”. In: *Proceedings of the 2011 Winter Simulation Conference (WSC)*. IEEE. 2011, pages 3947–3957.

[Stå99] I. Ståhl. *aGPSS*. http://agpss.com/index.html. 1999. Used in 2020.

[SV11] R. Shaikh and H. Vangheluwe. “Transforming UML2.0 Class Diagrams and Statecharts to Atomic DEVS”. In: *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. TMS-DEVS ’11. Boston, Massachusetts: Society for Computer Simulation International, 2011, pages 205–212.

[Syr+13] E. Syriani et al. “AToMPM: A Web-based Modeling Environment”. In: *Proceedings of MODELS’13 Demonstration Session*. 2013, pages 21–25.

[Syr11] E. Syriani. “A Multi-Paradigm Foundation for Model Transformation Language Engineering”. PhD thesis. McGill University, Feb. 4, 2011.

[SZ98] H. S. Sarjoughian and B. P. Zeigler. “DEVSJAVA: Basis for a DEVS-based collaborative M&S environment”. In: *Simulation Series* 30 (1998), pages 29–36.

[Tra09] M. K. Traoré. “A graphical notation for DEVS”. In: *Proceedings of the 2009 Spring Simulation Multiconference*. 2009, pages 1–7.

[Tuk77] J. W. Tukey. *Exploratory data analysis*. Volume 2. 1977.

[Uhr01] A. M. Uhrmacher. “Dynamic Structures in Modeling and Simulation: a Reflective Approach”. In: *ACM Transactions on Modeling and Computer Simulation* 11 (2001), pages 206–232.

[Uni05] Unity Technologies. *Unity*. https://unity.com/. 2005. Accessed 08-06-2020.

[Van+17] S. Van Mierlo et al. “Domain-specific modelling for human-computer interaction”. In: *The Handbook of Formal Methods in Human-Computer Interaction*. Springer, 2017, pages 435–463.

[Van00a] H. Vangheluwe. “DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling”. In: *IEEE International Symposium on Computer-Aided Control System Design*. Anchorage, AK, USA, 2000, pages 129–134.

[Van00b] H. Vangheluwe. “Multi-Formalism Modelling and Simulation”. PhD thesis. Universiteit Gent, 2000.

[Van08] H. Vangheluwe. “Foundations of modelling and simulation of complex systems”. In: *Electronic Communications of the EASST* 10 (2008).

[Van14] Y. Van Tendeloo. “Activity-aware DEVS simulation”. Master’s thesis. University of Antwerp, June 28, 2014.

[VV14] Y. Van Tendeloo and H. Vangheluwe. “The Modular Architecture of the Python(P)DEVS Simulation Kernel”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS*. Tampa, FL, USA, 2014, pages 97–102.

[VV15] Y. Van Tendeloo and H. Vangheluwe. “PythonPDEVS: a Distributed Parallel DEVS simulator”. In: *Proceedings of the 2015 Spring Simulation Multiconference*. Alexandria, VA, USA, 2015, pages 844–851.

[VV16] Y. Van Tendeloo and H. Vangheluwe. “An Overview of PythonPDEVS”. In: *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*. Toulouse, France: Cépaduès, 2016, pages 59–66.

[VV17a] Y. Van Tendeloo and H. Vangheluwe. “An Evaluation of DEVS Simulation Tools”. In: *Simulation* 93.2 (2017), pages 103–121.

[VV17b] Y. Van Tendeloo and H. Vangheluwe. “Classic DEVS Modelling and Simulation”. In: *Proceedings of the 2017 Winter Simulation Conference*. WSC 2017. Las Vegas, NV, USA: IEEE, Dec. 2017, pages 644–656.

[VV18] Y. Van Tendeloo and H. Vangheluwe. “Extending the DEVS Formalism with Initialization Information”. In: *ArXiv e-prints* (2018).

[VVD19] S. Van Mierlo, H. Vangheluwe, and J. Denil. “The Fundamentals of Domain-Specific Simulation Language Engineering”. In: *2019 Winter Simulation Conference (WSC)*. Dec. 8, 2019, pages 1482–1494.

[VVV18] S. Van Mierlo, Y. Van Tendeloo, and H. Vangheluwe. “A Generalized Stepping Semantics for Model Debugging”. In: *Proceedings of MODELS 2018 Satellite Events*. Volume 2245. CEUR-WS, Nov. 2018.

[Wai09] G. A. Wainer. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. 1st edition. Boca Raton, FL, USA: CRC Press, 2009.

[WK04] T. Williams and C. Kelly. *Gnuplot*. http://www.gnuplot.info/. 2004. Used in 2020.

[Zei84] B. P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[ZPK00] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. 2nd edition. Academic Press, 2000.

# Index

agent-based modeling, 132
atom, 8, 33

block diagram, 14
boxplot, 93–95, 99
building blocks, 2, 14, 15
  collectors, 64–69
    collector, 65, 66
    counter, 68, 69
    estimate collector, 67, 68, 95
    table collector, 64, 65
  generators, 41–54
    bulk generator, 54
    constant generator, 41, 42
    function generator, 42
    incrementor, 42
    random delay generator, 52
    random number generator, 51, 52
    rdg, 52
    rng, 51, 52
    single fire, 54
    stock, 53
    table generator, 42, 43
  input and output, 74–79
    file reader, 78
    file writer, 76, 77, 132
    listener, 78, 79
    loggers, 74–76
    sound, 78, 79, 132
math, 69–74
  adder, 70, 99
  differentiator, 72
  equation, 71
  integrator, 72
  multiplier, 70, 99
  random, 73, 74
queues, 55–64
  advance, 19, 63, 64
  queue, 60
  queue tracker, 61
  retain, 62
  simple queue, 60
routing, 82–101
  choose input, 85, 99
  choose output, 85, 86, 99
  delayer, 90, 91
  finish, 83
  gate, 89
  guard, 87–88
  halt, 83
  pick, 86, 99
  sync, 92, 99, 131
  timer, 90
transformers, 80–82
  lookup table, 81
  pack, 81, 82
  transformer, 80, 132
  unpack, 82

critical section, 19, 87

DEVS
- Atomic, 8–9
- bag, 10
- Closure under Coupling, 9–10
- closure under coupling, 10
- component references, 9
- confluent transition function, 10
- Coupled, 9
- external transition function, 8
- flattening, 9
- initial state, 8
- input set, 8, 9
- internal transition function, 8
- null event, 9
- output function, 9
- output set, 8, 9
- Parallel, 10
- passive state, 8
- select, 119
- set of all sub-components, 9
- set of influences, 9
- superdense time, 14
- tie-breaking function, 9, 10
- time advance function, 8, 105
- transfer function, 9

Enterprise Dynamics, 33
ExtendSim, 30

FlexSim, 31–33
flow architecture, 132
footprint, 95

GPSS, 14–28
- blocks
  - ADVANCE, 19, 20
  - ASSIGN, 19
  - DEPART, 28
  - ENTER, 22, 23
  - GATE, 22, 24
  - GENERATE, 17, 18
  - LEAVE, 22, 23
  - LINK, 23, 25
  - LOGIC, 22, 24
  - MARK, 27
  - PREEMPT, 20, 22
  - QUEUE, 28
  - RELEASE, 20, 21
  - RETURN, 21, 22
  - SEIZE, 20, 21
  - TABULATE, 25, 27
  - TERMINATE, 17, 18
  - TEST, 18, 20
  - TRANSFER, 18, 19
  - UNLINK, 23, 25
- chains, 16
- current events chain, 16
- delay chain, 16
- entity, 15
- facility, 19, 20
- future events chain, 16, 105
- interrupt chain, 16, 21
- logic switch, 19, 22
- match chain, 16
- non-mobile entities, 16
- scanning algorithm, 17, 21, 104, 106
- standard numerical attributes, 16, 104
- statements, 15
- storage, 19, 21
- termination counter, 17
- transactions, 15
- user chain, 16, 116

INCONTROL, 33

LEGO Mindstorms, 33–34

majorizing function, 49
meta-model, 4, 39
model transformation, 2

name mangling, 120

pull port, 13
push port, 13

queue, 55
  balking, 56, 62
  balking index, 57
  chunking, 58
  cutting lines, 58
  dequeue, 55
  discipline, 56
  enqueue, 55
  faffing, 58
  first-come-first-served, 56
  first-in-first-out, 28, 56
  jockeying, 57
  Kendall's notation, 55, 56
  last-in-first-out, 56
  priority, 56
  queueing theory, 55
  reneging, 57
  service-mechanism, 55
  skip, 58
  slip, 58

random, 43, 73
  acceptance-rejection, 49
  composition, 48
  convolution, 49
  goodness of fit, 50
  inverse-transform, 46
  LCG, 44
  linear congruantial generator, 44
  MCG, 44, 104
  multiplicative congruantial generator, 44, 104
  patternicity, 44
  predictiveness, 44
  pseudo-random, 44
  ratio of uniforms, 49
  reproducibility, 44
  seed, 51
  table method, 49
Random Number Generator, 43–52, 73

shift, 89
Simpson's rule, 73
SIMUL8, 31
state set, 8
stock, 52
syncing, 91
syslog, 74

termination, 83
timeframe, 88, 91
tracers
  Footprint Tracer, 95
  Plot Tracer, 93
  Statistics Tracer, 95
transformers
  assembling, 81
  batching, 81
  combining, 81
  packing, 81
  separating, 81
  unbatching, 81
  unpacking, 81