Model Oriented Specification of Communicating Agents
Model Oriented Specification of Communicating Agents

Proefschrift

Ter verkrijging van de graad van doctor aan de Technische Universiteit Delft
op gezag van de Rector Magnificus,
Prof. Drs. P.A. Schenck,
in het openbaar te verdedigen
ten overstaan van een commissie
aangewezen door het College van Dekanen
op vrijdag 18 december 1992 te 19.00 uur

door

Wilhelmus Johannes Toetenel

geboren te Den Haag,
wiskundig ingenieur.
Dit proefschrift is goedgekeurd door de promotor:

prof. dr. ir. J. van Katwijk
To Patty, Rik, Ben and Thijs
# Contents

1 Introduction .......................... 1
  1.1 Computing systems .................. 1
  1.2 Modeling and analysis ............... 2
  1.3 Timed simultaneous computation ..... 4
  1.4 The research ....................... 6
    1.4.1 Background ..................... 6
    1.4.2 Subject of Research and Hypothesis 6
    1.4.3 Research assessment ............. 8
  1.5 Thesis overview .................... 8

2 Context ................................ 11
  2.1 Introduction ......................... 11
  2.2 The Vienna Development Method ..... 12
    2.2.1 Introduction and overview ..... 12
    2.2.2 The VDM-SL notation .......... 15
  2.3 Process specification ............... 17
    2.3.1 Introduction ................... 17
    2.3.2 Process Algebras ............... 18
    2.3.3 CCS ........................... 19
  2.4 Time ................................ 23
  2.5 Combination notations .............. 26
    2.5.1 Value and Process combinations 26
    2.5.2 Value and Time combinations ... 29
    2.5.3 Process and Time combinations 30
    2.5.4 Value, Process and Time combinations 38

3 Notation Overview .................... 39
  3.1 Introduction ......................... 39
  3.2 Agent definition .................... 40
    3.2.1 Type 0 Agents .................. 41
## CONTENTS

3.2.2 Type I Agents .................. 42
3.2.3 Type II Agents ................. 47
3.2.4 Mixed Agents ................ 50
3.3 Agent Behaviour ................ 50
3.3.1 Instantiation and Termination 51
3.3.2 Composition, Relabelling and Restriction 52
3.4 Units, Collections, Scopes and Visibility 56
3.4.1 Units .......................... 56
3.4.2 Collections .................. 58
3.4.3 Scope and Visibility .......... 60
3.5 Time specification ............... 62
3.5.1 Overview .................... 63
3.5.2 Timed Prefix .................. 65
3.5.3 Idle Action ................... 66
3.5.4 Timed Actions ................ 66
3.5.5 Examples ..................... 67
3.6 Summary .......................... 71

4 Semantics 75
4.1 Loose process semantics .......... 75
4.2 Overview of the semantics of MOSCA 80
4.3 VDM-SL Semantics ............... 82
  4.3.1 The universe of Complete Partial Orders 84
  4.3.2 Basic Semantic Domains .......... 85
  4.3.3 Overall Strategy ............... 87
  4.3.4 The evaluation functions and domains 89
4.4 MOSCA's transition system ....... 89
  4.4.1 The environment ............... 91
  4.4.2 The action set Act ............ 94
  4.4.3 The transition relation .......... 95
  4.4.4 State manipulation .......... 99
  4.4.5 The interface function VS ..... 100
  4.4.6 Environment manipulation .... 101
4.5 Semantics of Time ............... 105
  4.5.1 Basic Properties ............. 105
  4.5.2 Semantic rules ............... 106
4.6 Semantics of looseness .......... 107
  4.6.1 Loose value specification ..... 107
  4.6.2 Loose Agent specification .... 111
  4.6.3 Loose Time Specification ...... 118
CONTENTS

5 Development
  5.1 Introduction ................................................. 121
  5.2 On finding the intended class of systems .................. 122
  5.3 Software development ........................................... 124
    5.3.1 A simple software life cycle model ...................... 124
    5.3.2 Software Development Activities ......................... 125
    5.3.3 Models in the SDA ......................................... 127
  5.4 The models of VDM, CCS and TIME ......................... 131
    5.4.1 VDM in software development ......................... 131
    5.4.2 CCS in software development ......................... 132
    5.4.3 Time in software development ......................... 137
  5.5 MOSCAin software development ............................... 139

6 Verification .................................................. 143
  6.1 Introduction ................................................. 143
  6.2 State space analysis ........................................... 143
    6.2.1 Introduction ............................................. 143
    6.2.2 The total and derived state spaces ..................... 145
    6.2.3 The state space classes $SS^F_r$, $SS^R_r$ and $SS^D_r$ .... 146
    6.2.4 The state spaces induced by restriction ................. 150
    6.2.5 Properties of state spaces ................................ 155
    6.2.6 State space generation ................................... 161
  6.3 Trace analysis ................................................ 164
    6.3.1 Introduction .............................................. 164
    6.3.2 Finite Timed traces ...................................... 165
    6.3.3 Some remarks concerning trace generation ............... 169
    6.3.4 Traces of agents with cyclic behaviour ................. 172
    6.3.5 Analysis by traces ...................................... 173
  6.4 Conclusion .................................................. 178
    6.4.1 State spaces in analysis ................................ 178
    6.4.2 Traces in analysis ...................................... 180
    6.4.3 Other approaches ....................................... 180

7 Application ............................................... 183
  7.1 Introduction ................................................. 183
  7.2 A communication protocol ..................................... 183
    7.2.1 Introduction ............................................. 184
    7.2.2 Two-Way Transmission Medium ............................. 185
    7.2.3 An abstract $2WTM$ model ................................ 186
    7.2.4 A concrete model of the $2TWB$ ......................... 190
    7.2.5 Analysis of $2WTM$ ...................................... 201
CONTENTS

7.2.6 Conclusion ........................................ 210
7.3 Process control systems ............................... 212
  7.3.1 Introduction .................................. 212
  7.3.2 The heater ..................................... 213
  7.3.3 The heater process ............................. 216
  7.3.4 The control component ......................... 226
  7.3.5 Analysis ....................................... 237
  7.3.6 Discussion ................................... 245

8 Implementation ........................................ 249
  8.1 Introduction .................................... 249
  8.2 Terminology ...................................... 251
  8.3 The P2L3S Protocol ................................. 256
  8.4 P2I3S scenarios .................................. 263
  8.5 P2I3S analysis .................................. 270
  8.6 Extensions to the protocol ....................... 282
  8.7 Related work .................................... 284
  8.8 Conclusions ..................................... 285

9 Evaluation ............................................ 287
  9.1 Introduction .................................... 287
  9.2 Validation of the research effort ............... 287
    9.2.1 Validation of MOSCA ......................... 287
    9.2.2 The research hypothesis revisited .......... 292
  9.3 Future research .................................. 293
    9.3.1 Matters concerning the language .......... 293
    9.3.2 Matters concerning specification .......... 295
    9.3.3 Matters concerning analysis ............... 296
    9.3.4 Matters concerning implementation .......... 296
  9.4 To conclude ..................................... 296

A Presentation syntax .................................. 299
  A.1 Introduction .................................... 299
  A.2 Collection Specification ......................... 301
  A.3 Agent Specification .............................. 302
    A.3.1 Agent Definition ............................ 302
    A.3.2 Agent Behaviour ..................................... 304
  A.4 Value Specification .................................. 305
## CONTENTS

**B Core Abstract Syntax**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Introduction</td>
<td>307</td>
</tr>
<tr>
<td>B.2 Collection Specification</td>
<td>307</td>
</tr>
<tr>
<td>B.3 Agent Specification</td>
<td>308</td>
</tr>
<tr>
<td>B.3.1 Agent Definition</td>
<td>308</td>
</tr>
<tr>
<td>B.3.2 Agent Behaviour</td>
<td>309</td>
</tr>
<tr>
<td>B.3.3 Auxiliary Domains</td>
<td>310</td>
</tr>
</tbody>
</table>

**C Semantic Rules**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1 Loose environment manipulation rules</td>
<td>312</td>
</tr>
<tr>
<td>C.2 Loose action rules</td>
<td>320</td>
</tr>
<tr>
<td>C.2.1 Conditional expression</td>
<td>320</td>
</tr>
<tr>
<td>C.2.2 Prefix expressions</td>
<td>321</td>
</tr>
<tr>
<td>C.3 Standard action rules</td>
<td>322</td>
</tr>
<tr>
<td>C.3.1 Prefix expression</td>
<td>322</td>
</tr>
<tr>
<td>C.3.2 Choice expressions</td>
<td>322</td>
</tr>
<tr>
<td>C.3.3 Composition expressions</td>
<td>323</td>
</tr>
<tr>
<td>C.3.4 Restriction and relabelling expressions</td>
<td>325</td>
</tr>
<tr>
<td>C.4 Idling rules</td>
<td>326</td>
</tr>
<tr>
<td>C.5 Null agents and Divergence</td>
<td>330</td>
</tr>
</tbody>
</table>

**Bibliography**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>351</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Main VDM Dialects</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>CCS Structured Operational Semantics</td>
<td>21</td>
</tr>
<tr>
<td>2.3</td>
<td>Comparison of Timed CCS approaches - 1</td>
<td>32</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of Timed CCS approaches - 2</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Agent / Value combinations</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Typical agent pictorials</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>MOSCA Behaviour expressions</td>
<td>51</td>
</tr>
<tr>
<td>3.4</td>
<td>Shared buffer with two clients</td>
<td>57</td>
</tr>
<tr>
<td>3.5</td>
<td>Nested Scope levels</td>
<td>61</td>
</tr>
<tr>
<td>3.6</td>
<td>Mosca features summery</td>
<td>73</td>
</tr>
<tr>
<td>4.1</td>
<td>Semantic Structure of LOOP</td>
<td>81</td>
</tr>
<tr>
<td>4.2</td>
<td>Semantic Structure of MOSCA</td>
<td>83</td>
</tr>
<tr>
<td>4.3</td>
<td>Environment manipulation within inference chains</td>
<td>102</td>
</tr>
<tr>
<td>4.4</td>
<td>Derivation graph for agent $Z$</td>
<td>113</td>
</tr>
<tr>
<td>4.5</td>
<td>Pictorial for the body of agent $System$</td>
<td>117</td>
</tr>
<tr>
<td>4.6</td>
<td>Derivation graph with symbolic sts values for $System$</td>
<td>118</td>
</tr>
<tr>
<td>5.1</td>
<td>Correspondance between language features and systems</td>
<td>123</td>
</tr>
<tr>
<td>5.2</td>
<td>Software Development Activities</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>VDM and CCS in the SDA</td>
<td>139</td>
</tr>
<tr>
<td>5.4</td>
<td>Coverage of VDM and CCS</td>
<td>140</td>
</tr>
<tr>
<td>6.1</td>
<td>The total and derived state spaces</td>
<td>147</td>
</tr>
<tr>
<td>6.2</td>
<td>Non-deterministic graph $TSS$ of $P$</td>
<td>150</td>
</tr>
<tr>
<td>6.3</td>
<td>Deterministic graph $SS^R$ of $P$</td>
<td>150</td>
</tr>
<tr>
<td>6.4</td>
<td>Partially enfolded full complete graph of $ticker$ (3)</td>
<td>152</td>
</tr>
<tr>
<td>6.5</td>
<td>Reduced complete graph of $ticker$ (3)</td>
<td>153</td>
</tr>
<tr>
<td>6.6</td>
<td>Partially enfolded complete graph of $IO$ (0) $</td>
<td>$ $Clock$ (0)</td>
</tr>
<tr>
<td>6.7</td>
<td>Partially enfolded $SS^F_\xi$ graph of $IOC$</td>
<td>157</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.8</td>
<td>$SS^D_e$ graph of $IOC$</td>
<td>157</td>
</tr>
<tr>
<td>6.9</td>
<td>Preservation of space structure</td>
<td>160</td>
</tr>
<tr>
<td>6.10</td>
<td>Trace classes</td>
<td>166</td>
</tr>
<tr>
<td>6.11</td>
<td>Finite Timed Traces</td>
<td>168</td>
</tr>
<tr>
<td>6.12</td>
<td>The TMR system</td>
<td>174</td>
</tr>
<tr>
<td>7.1</td>
<td>Communication Protocols</td>
<td>185</td>
</tr>
<tr>
<td>7.2</td>
<td>A simplified Sender/Receiver Connection Scheme</td>
<td>191</td>
</tr>
<tr>
<td>7.3</td>
<td>Sender-First Scenario</td>
<td>192</td>
</tr>
<tr>
<td>7.4</td>
<td>Reader-First Scenario</td>
<td>194</td>
</tr>
<tr>
<td>7.5</td>
<td>Two-way-buffer states</td>
<td>196</td>
</tr>
<tr>
<td>7.6</td>
<td>Structure of the $2WTM$ specification</td>
<td>197</td>
</tr>
<tr>
<td>7.7</td>
<td>Total $mn$ state space abstract model</td>
<td>204</td>
</tr>
<tr>
<td>7.8</td>
<td>Reduced total state space abstract model</td>
<td>205</td>
</tr>
<tr>
<td>7.9</td>
<td>Reduced observational state space refined model</td>
<td>207</td>
</tr>
<tr>
<td>7.10</td>
<td>Basic model of a process and its control system</td>
<td>213</td>
</tr>
<tr>
<td>7.11</td>
<td>Informal requirements of Heating System</td>
<td>214</td>
</tr>
<tr>
<td>7.12</td>
<td>Basic model of the process part</td>
<td>216</td>
</tr>
<tr>
<td>7.13</td>
<td>Main switch push-button model</td>
<td>219</td>
</tr>
<tr>
<td>7.14</td>
<td>Main switch tumble model</td>
<td>220</td>
</tr>
<tr>
<td>7.15</td>
<td>Basic model of the control part</td>
<td>227</td>
</tr>
<tr>
<td>7.16</td>
<td>Continuation chart process actuators and sensors</td>
<td>238</td>
</tr>
<tr>
<td>7.17</td>
<td>Continuation chart interface devices</td>
<td>239</td>
</tr>
<tr>
<td>7.18</td>
<td>Continuation chart controller</td>
<td>240</td>
</tr>
<tr>
<td>8.1</td>
<td>Parallel Synchronizations</td>
<td>250</td>
</tr>
<tr>
<td>8.2</td>
<td>Binary Connection Network Tree</td>
<td>252</td>
</tr>
<tr>
<td>8.3</td>
<td>Action structure value state changes</td>
<td>254</td>
</tr>
<tr>
<td>8.4</td>
<td>Simple synchronization between two parties</td>
<td>255</td>
</tr>
<tr>
<td>8.5</td>
<td>P2L3S Phases State Diagram</td>
<td>256</td>
</tr>
<tr>
<td>8.6</td>
<td>Three parties: a 2 — 3 synchronization</td>
<td>264</td>
</tr>
<tr>
<td>8.7</td>
<td>Three parties: a 1 — 3 synchronization</td>
<td>265</td>
</tr>
<tr>
<td>8.8</td>
<td>Delayed pebbling</td>
<td>266</td>
</tr>
<tr>
<td>8.9</td>
<td>Pebbling Resumed</td>
<td>267</td>
</tr>
<tr>
<td>8.10</td>
<td>Repebbling 1</td>
<td>268</td>
</tr>
<tr>
<td>8.11</td>
<td>Repebbling 2</td>
<td>269</td>
</tr>
<tr>
<td>8.12</td>
<td>Postponed pebblers on pivot</td>
<td>270</td>
</tr>
<tr>
<td>8.13</td>
<td>Trees with depth 0 to 3</td>
<td>273</td>
</tr>
<tr>
<td>8.14</td>
<td>Atu's in Synchronizations</td>
<td>277</td>
</tr>
<tr>
<td>8.15</td>
<td>Cycling through states 3—4—9</td>
<td>278</td>
</tr>
<tr>
<td>8.16</td>
<td>Possible Synchronizations in $T(p)$</td>
<td>279</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

9.1 Enfolding mix agents ........................................ 295
Words of Thanks

My first words of thanks are to my colleague, tutor and promotor, professor Jan van Katwijk. He has guided my ongoing struggle to define, refine and finalize the MOSCA language. Without ever being restrictive he has offered me the means to complete this work, and provided me with his deep insight to keep this work firmly based in the discipline of software engineering. The presentation of this thesis in its current form had never been accomplished without his substantial comments and ideas for improvements. Thank you very much Jan!

This research has been inspired by the work of many scientists.
Foremost I am indebted to professor Dines Bjørner who introduced me to the syntactic, semantic and pragmatic aspects of VDM-SL. He provided me initially with the necessary documents that covered these aspects and inspired me first of all by his tutorial presentation given at the 1990 VDM-Europe conference in Dublin where he distributed preprints of his vast, almost encyclopedic work on the foundations and application of VDM. Later he provided deep insights covering various aspects of the work under progress. Without his aid and understanding this work had never been accomplished. Second in my list of scientists to whom I am deeply indebted is professor Robin Milner. Without his extremely important work on CCS I had never been able to conceive the topic of this thesis.

Next I wish to express my gratitude to the members of the VDM-Europe community that in one way or other contributed to the topic of this thesis. In this context I have to mention personally professors Cliff Jones, Micheál Mac an Airchinnigh, Hans Langmaack and Kees Middelburg. They have inspired me deeply with their scientific work.

I am very glad to have been able to work directly with Peter Larsen and Nico Plat. They have been very helpful with their comments and suggestions concerning the various topics of this work.
My special thanks are directed to Arlet Ottens, a graduate student who assisted me with the implementation of the MOSCA rapid prototyping tool. He implemented a first prototype of the tool and provided many suggestions concerning inconsistencies in the MOSCA language.

Thanks to all my colleagues in Delft that contributed in one way or the other to this work, in particular Mrs. Trudie Stoute who tried (in vain) to improve my English style and vocabulary.

Thanks to my parents Leidy and Pim Toetenel, who first created a fine place to live and study and later offered their support during difficult times. Thanks to my parents-in-law Heilke and Herman Hasz who were always ready to stand by to let their son-in-law struggle on.

But my last words of thanks are to my wife Patty and children Rik, Ben and Thijs. I thank them for their patience, understanding, endurance and support during the long period their husband and father was in other worlds. But now he is back on Earth.
1.1 Computing systems

In the last two decades many techniques and methodologies were developed that have as common aim the systematic and orderly development of computer software. With the emerging techniques came the recognition that the development of software was not one huge homogeneous activity, but could be split up in the different phases of the software life cycle model. In this model the actual coding of the software is preceded by careful analysis of the problem for which an automated solution is sought. The analysis results in a description of a solution which after thorough design leads to the desired software product.

There is nowadays a problem concerning the early techniques of the 1970s and 1980s. Although often very convincing and elusive through their various diagrammatic presentations, they lack formality, and are in a sense informal techniques. As the complexity of the systems that are developed now has increased enormously in comparison with the complexity of the systems that were developed in the 1970s and 1980s the need for formal verification and simulation based on formal notations has become even more important.

Research over the last 20 years has developed formal techniques for the development of sequential, or transformational computing systems. These techniques are now quite well accepted within the research community and are adopted in many industrial applications. Transformational systems can be characterized as computation oriented systems having strictly limited and well guarded interaction with their environment, as opposed to a completely different class of systems that can be characterized as computer systems with an ongoing interaction with their environment.
Chapter 1. Introduction

Unfortunately these latter systems cannot be sensibly viewed within the transformational framework. Most of these systems are highly concurrent, distributed and often have real-time properties. The development of these systems is far more complex than the development of data driven systems. Many authors like e.g. Brooks [44], Parnas [174], and Leveson [140] have stressed that the current standard techniques used in the development of data driven systems do not match the demands put on these development techniques that result from the properties of real-time and distributed systems. Even worse, the trial of building such systems using the same approaches developed and used for data processing and information processing systems will lead to grave failures.

The problems are different and require different techniques. For example, Leveson argues in [140] that modelling and analysis form the main challenges in building real-time (control) systems. Stankovic offers similar observations in [199]. He states that the main challenge in the development of advanced computing systems lies in the modelling and verification of timing constraints. The inclusion of a time metric often increases the semantics of concurrency models to a great extend, and thus complicate the verification problem. One of the more difficult problems is caused by the fact that the verification of these systems requires the satisfaction of timing constraints, where those constraints are derived from the environment and the implementation.

The verification methods normally abstract away from the implementation even though it is the implementation and the environment that provide the true timing constraints.

1.2 Modeling and analysis

Advanced digital computing systems encompass many different qualities, such as parallel-, embedded-, distributed-, concurrent-, process-control- and real-time systems. Although some of these terms seem to be orthogonal, it is not an easy task to define the specific characterizations for these kind of systems. More revealing are some of the basic properties of these systems which I would like to bundle together in the next definition.

Definition 1.1 Timed Simultaneous Computing Systems A timed simultaneous computing digital system, further referenced by the acronym TSCS, consists in general of a collection of software and hardware components, interacting in a complex manner. The behaviour of the system that is determined by computations is in general not only dependent on the logical result of the computation, but also depending on the moment
1.2. Modeling and analysis

in time the result is produced.

There are many different techniques and notations to model and specify timed simultaneous computations within TSCS's. These techniques and notations range from highly abstract to very concrete. Some are formal in the sense that they are based on mathematics and offer a means to reason about certain properties of the specification. Others are structured in the sense that they apply well defined rules to create informal specifications out of well defined building bricks. Again others are graphical, tabular-oriented, etcetera. Mixed forms are also applied regularly.

Modeling and analysis are important issues in the development of TCSC's. These two activities can be applied on three different kinds of system properties: functionality temporal behaviour and real-time behaviour.

- The functional capabilities of software components can be formally stated with formal specification techniques developed over the last 20 years, like model-oriented or property oriented techniques. Model-oriented techniques encompass VDM and Z, property-oriented methods encompass ACT-ONE, OBJ, LARCH etc.

- The temporal behaviour capabilities of software components are often divided into the now well known classes of liveness and safety properties, as proposed by Owicki and Lamport in [172]. These properties state invariant properties over the whole lifetime of a system. The temporal behaviour capabilities can be expressed by temporal logics, process based notations and calculi like CCS and CSP, Petri-Net specifications and finite-state-automata.

- The real-time behaviour capabilities of software components describe properties that are valid only on intervals of the lifetime of a system or are bound to explicit moments in time. The real-time behaviour capabilities can be divided into two classes: requirements of responsiveness of events and requirements of timing constraints. The responsive requirement states merely a temporal relationship between two events e and f, without stating when e or f should occur or what their durations must be. The timing constraints do expresses these latter properties. There are e.g. real-time temporal logics, real-time logics, real-time process based notations, Petri-Nets with real-time extensions and finite-state-automata with real-time extensions.

Techniques dedicated to one of these three classes can be combined to cover all aspects of the software components of TSCS's.
These techniques and notations can be applied on various levels of abstraction, corresponding with different activities in the life cycle model. Most of these techniques offer the potential of formal verification of certain system properties. Some of them enable dynamic verification by simulation or rapid prototyping.

1.3 Timed simultaneous computation

There is at present not one specific notation that is widely accepted for modelling of systems that perform computations simultaneously. There is instead a proliferation of semantics for such languages. Some use widely applicable methods and are based on, for example Petri-Nets and labelled transition systems, while some are more specialised and are designed with an idea of detecting and proving specific properties of specifications.

The computations within a specification is often denoted by the concept action. Without giving a precise definition of the concept it will be understood as a single input action or output action or an internal action corresponding with a single computation step not involving any input or output. The actions in a specification can be ordered by a relation such as "is directly followed by". Let $A$ denote the total set of actions within a specification. Let $\langle \subset A \times A$ denote the relation "is directly followed by". Together with the dual relation $\rangle$ meaning "is directly caused by" a precise characterization can be given of the succession of actions within a specification. Both relations are irreflexive, asymmetric and transitive, thus strict total ordering relations (inhibiting circular dependencies). A small example is given to illustrate this point. Suppose $A$ contains the symbolically named actions $a, b, c, d, e$ with associated computations (a) input the numbers $p$ and $q$, (b) print the numbers on a visual display unit, (c) compute the value $x = p^2$, (d) compute the value $y = 4pq$, (e) compute the value $x - y$. So, the relation $\langle$ consists of $\{(a, b), (a, c), (a, d), (c, e), (d, e)\}$. This relation can be visualised by its graph.

Such a graphical representation of an action ordering can be used as structuring mechanism in a specific semantic model for the specification. This approach is taken amongst others in net-based specifications like Petri-Nets.
1.3. **Timed simultaneous computation**

in all its variants, and state-machine-based specifications. The semantic model then offers a meaning to communication and concurrency aspects within the structure. I would like to characterize these models as: *models that specify single actions linked by causality.*

Another class of semantic models is based on action sequences. As usual I let $\prec^{*}$ denote the transitive closure of $\prec$. The next functions computes the elements in the transitive closure related to a special element $a$.

\[ \prec : A \rightarrow A \text{-set} \]
\[ \prec(a) \triangleq \{ a_j \mid a \in A \cdot a \prec^{*} a_j \} \]

\[ \succ : A \rightarrow A \text{-set} \]
\[ \succ(a) \triangleq \{ a_j \mid a \in A \cdot a \succ^{*} a_j \} \]

Let $\prec$ and $\succ$ be the functions defined as

\[ \prec (a; A) r; A \]
\[ \text{post } r \in \prec(a) \land \neg \exists e \in \prec(a) \setminus \{ a \} \cdot a < e \]

\[ \succ (a; A) r; A \]
\[ \text{post } r \in \succ(a) \land \neg \exists e \in \succ(a) \setminus \{ a \} \cdot a > e \]

By means of these functions starting actions that have no predecessors can be found, and similarly ending actions that have no successors. The actions that are executed in a sequence from a starting action to an ending action chained together form a *thread of control* which are generally embedded in a concept denoted as *process*. Processes as such have starting actions and ending actions and consist of related actions which are executed sequentially.

Processes can be specified in a various number of styles. For example a rather strict style that describes the relation $\prec$ almost literally is the *actor* approach of *Hewitt* and *Aga* ([7]). A more algebraic style is found in *Hoare's CSP* and *Milner's CCS*, both extensively documented in their books [103] and [157]. Even pure algebraic specification models exist, like for instance the ACP approach of *Bergstra* and *Baeten* [28].

As the importance to address timing related topics grew more and more, many of the notations dedicated to simultaneous computing became equipped with some form of time registration. Thus evolved time(d) Petri-Nets, timed finite state machines, timed transition systems and variants of CSP and CCS with time extensions.
1.4 The research

1.4.1 Background

The research described in this thesis was conceived during a period in which I have been strongly occupied with the VDM-SL, the specification notation of the Vienna Development Method. Before getting engaged in the study of various aspects of this specification language my primary research interest has been with programming languages and their implementation techniques. I have been involved in a project dedicated to the programming language Ada and its implementation models ([122]). Through the study of Ada and its implementation models I became acquainted with formal methods to describe concurrency, and developed a special fondness for the calculus of communicating systems, CCS, developed by Robin Milner. CCS consists of a terse set of primitives and has been given both an algebraic characterization as an operational semantic characterization. CCS has had an enormous impact on the development of theories of concurrency. However, my interest in the notation was oriented on the more practical aspects of *application* of the notation.

The occupation with VDM-SL came forth from a growing interest in the fundamentals of programming and programming notations. My research interests that grew from the Ada project were strongly biased towards the specification of programs. During the project we have worked amongst others on the design and implementation of an Ada runtime kernel to support Ada tasking ([124], [125], [212]). The resulting runtime kernel proved to be both efficient and elegant. To capture its essential model we set out to provide a formal description of the implementation. Although this experiment never saw its conclusion it gave me the context to study VDM-SL.

Through our study of VDM-SL ([207], [178], [176], [177]) and its associated method(s) for application we came acquainted with its formal methods society, VDM-Europe,¹ through which we became involved with the standardisation of the notation ([179]). The involvement culminated in the organisation of the fourth international VDM symposium in 1991 ([185], [186]).

1.4.2 Subject of Research and Hypothesis

During the study of VDM-SL and the the vast class of notations, paradigms and approaches mentioned earlier to describe concurrency I retained my strong interest in the CCS notation of Robin Milner.

¹Now, FME, Formal Methods Europe.
1.4. The research

As VDM-SL lacks any construction to specify concurrency and CCS lacks a well defined form for data specification and manipulation I speculated that a combination of the two notations would produce a specification language that would combine the main properties of both constituents: data specification and specification of communication and concurrency. During a preliminary research to find publications on earlier similar experiments I found many extensions of CCS to enable the description of time dependent behaviour.

As most of these extensions were given a theoretically sound and complete formal semantics I set out to find a procedure to create a combination of VDM-SL and CCS that incorporated the facilities to describe time dependent behaviour as well.

The combination effort confronted me with issues associated with three different aspects of languages: syntactic aspects, semantic aspects and pragmatic aspects. Each different class represented specific problems.

- The syntactic level represented issues w.r.t. the syntactic representation of the different combined capabilities of the notation.

- The semantic issues were related to the choice of semantical base for the combination notation.

- On the pragmatic level the issues revolved around the choice of constructions that were incorporated in the language. E.g should it be desirable to have constructions for a specific purpose like exception handling be present for the three capabilities in the combination, i.e. for functional behaviour, temporal behaviour and real-time behaviour. VDM-SL was equipped with an exception handling mechanism, but the CCS extensions lacked such constructions.

Besides the language aspects there is an even more important aspect to consider. To what purpose was the combination effort directed? Possible answers on this question might include:

1. to deepen the understanding of theoretical aspects of specification languages, or

2. to establish a practical basis for specification of TSCS’s.

Initially my interests were dedicated to the second aspect, but it appeared inevitable to address issues related to the first aspect as well.

But finally the most important issue raised by the research quest may be formulated as follows:
Does the combination of the two notations VDM-SL and CCS make any sense at all?

Without a positive answer on this fundamental question the combination effort would be without any purpose at all. Thus the affirmative answer on this fundamental question is taken as the research hypothesis.

1.4.3 Research assessment

To judge the outcome of the research effort I have setup a small validation suite that addresses more or less related properties, which are:

1. Is the combination notation based on a complete and sound theoretical base?

2. Has the combination notation retained the important properties of the constituent parts?

3. Is the combination notation a possible candidate for fruitful application within the activities for TSCS development? Related issues to this topic are:

   (a) Is the language expressive enough to specify software components of TSCS’s?

   (b) Does the language support formal reasoning necessary for the validation of software components of TSCS’s?

   (c) Does the language preserve the implementation freedom needed to meet the nonfunctional requirements of the software components of TSCS’s?

   (d) Is there a method for construction, validation and implementation of specifications in the language?

   (e) Is the use of the language cost-effective for the construction of software components of TSCS’s? ²

1.5 Thesis overview

In the following chapters I will present the necessary background material, language definition, analysis tools and application case studies that will contribute to the answering of the questions stated above.

²The last 5 topics are directly inspired by the observations of Pamela Zave ([226]). She has conducted a research effort aimed at the construction of a software requirements specification language and its supporting toolset for TSCS’s for over nine years that resulted in the language PAISLey.
1.5. Thesis overview

The second chapter provides some essential background information on VDM-SL, CCS and some issues related to time. It addresses indirectly both the hypothesis and the second topic in the validation suite.

The next two chapters present the combination notation in detail. The notation is denoted by the acronym MOSCA, which is derived from the initial characters from the title of this thesis. These chapters cover the first topic in the validation suite. Chapter 3 presents an overview of the language on a syntactic and pragmatic level. In chapter 4 the semantic level is investigated and presents an operational semantics for the combination language. First the operational semantics of very small highly artificial language is presented that covers some of the basic properties of MOSCA, being loose value manipulation, and communication. This presentation introduces the following explanation of the semantics given to MOSCA.

In chapter 5 a simple framework is presented that addresses three important activities in the development of software for TSCS's: specification, implementation and verification. A taxonomy of desired properties is presented for notations directed at the support of these different activities. The question with respect to the hypothesis of the research cannot be resolved without first deciding on the particular qualities of both VDM-SL and CCS with respect to software development in general, which are covered in this chapter.

The next three chapters return to the question concerning the applicability of the combination language by engaging the combination notation in the three activities of specification, implementation and verification. Chapter 6 investigates properties related to the analysis of MOSCA specifications. Chapter 7 contains two detailed case studies of the application of MOSCA and chapter 8 addresses one particular aspect of implementing MOSCA specifications: the realization of the CCS primitives in the context of distributed applications.

Finally chapter 9 discusses the outcome of the research. It addresses the research hypothesis in the context of the presented material in the former chapters and formulates answers to the topics raised in the validation suite. Further it provides pointers into future research.
2.1 Introduction

In this chapter I present the relevant context information on the ingredients for the MOSCA combination, which are the VDM-SL notation, CCS, and a set of constructions to specify time-dependent behaviour.

The chapter opens with a section devoted to the Vienna Development method. I present a brief history and summary of the main VDM-SL dialects and a brief overview of the main constituents of the specification language. The semantic model of VDM-SL and its formal specification is deferred to be presented in chapter C, in which the semantic model of the MOSCA notation is treated in detail. The presentation in this section is partly based on [178].

The CCS approach is discussed in section 2.3. The presentation is restricted to basic CCS, i.e. CCS without value passing actions. The section opens with an introduction to process algebras in general. Both syntax and semantics of CCS are treated next and some tools for building equivalences and congruences are defined.

Traditionally, real-time systems and their software has been considered to be closely connected to time and its properties. Quite often the presence of explicit time in a computer system has been considered the main property of real-time systems. The notion of time seems to be a suitable tool to connect the software components of real-time systems with the rest of the components in the systems. Section 2.4 introduces some basic notations to describe the effect of time in specifications.

The chapter is concluded with an overview of combination notations that address constructions for data, process and time manipulation. The first topic of combinations addresses joint value and process combinations.
The second group of combination notations are related to the combination of data and time description. The most relevant combination is investigated next, the combination of process and time description techniques.

2.2 The Vienna Development Method

2.2.1 Introduction and overview

The Vienna Development Method (VDM for short) is a formal method for the specification and subsequent development of sequential, single threaded, state based software components. It is a model-oriented method; i.e. a VDM specification consists of an explicit model of the software component being constructed. VDM consists of two major components:

1. The VDM Specification Language, or VDM-SL for short. The language describes a notation in which VDM specifications can be expressed.

2. A method describing the development from a specification to an implementation in a certain programming language.

A VDM specification expressed in VDM-SL consists of a state of the system being constructed, and a number of operations that manipulate (parts of) this state. Auxiliary constructs are (mathematical) functions, type definitions and constant definitions.

A lot of constructions present in VDM-SL resemble similar constructions of most modern sequential programming languages nowadays. Exceptions are those constructions that contribute to the power of expressing abstraction, e.g. implicit function and operation specifications, maps and implicit set expressions. It is because of such constructs that VDM-SL is a specification language and not a programming language, and that VDM-SL specifications in general are not executable.

Whereas VDM-SL is rather well-defined, the method-part of VDM is not. The method consists of a series of development steps that transform a VDM specification into an implementation and of a number of proof obligations that must be performed during this development. The philosophy behind VDM assumes the presence of an initial specification at a high level of abstraction, which is refined in an arbitrary number of steps into an implementation. The result of each step is a new VDM specification, which is more concrete (i.e. contains more implementation details) than the previous one. Initially, attention is focused on the refinement of the data structures used in the specification, including the adjustment of the operations and
functions that use these data structures. This activity finally results in a specification in which data structures are used that closely resemble data structures present in the target programming language. After that the operations and functions in the specification are refined, i.e. implicit operations and functions are transformed into their explicit equivalents. Each step taken also results in a number of proof obligations which depend on the kind of step taken. The general purpose of the proof obligations is to prove that the step taken was correct. The proof obligations are described in detail in [115]. Some more remarks on the meaning of ‘M’ in VDM are made in [32].

Thus, although the method part of VDM confronts the user with the consequences of steps taken during the development process (i.e. proof obligations), there are no guidelines as to how the development steps should be taken. It is only suggested that the level of abstraction should be lowered, but VDM provides no guidelines to reach these lower abstraction levels.

VDM as such is a generic term. The development of VDM has given rise to several VDM ‘dialects’. The development of VDM started in 1970, when in the IBM laboratory in Vienna a group brought to Vienna by Heinz Zemanek worked on formal language definition and compiler design. They built on ideas of Elgot, Landin and McCarthy to create an operational semantics approach capable of defining the whole of PL/I including the parallel features of the language. For this purpose a meta-language which was called Vienna Definition Language (VDL) was used. The approach taken was in a sense successful, but it also showed that operational semantics could complicate formal reasoning in an unnecessary way. A new approach was taken, denotational semantics, in late 1972. A PL/I compiler was designed using a meta-language called Meta-IV. Due to external reasons the work on the compiler was never finished, but the formal definition of PL/I in a denotational style is generally seen as the birth of VDM [25].

The diversion of the IBM group to handle more practical problems led to its dissolution. From then on, further development took mainly place in two places: in Copenhagen (Prof. Dines Bjørner) and in Manchester (Prof. Cliff B. Jones). The Danish VDM research has concentrated on language/compiler problems, which has led e.g. to a complete formal definition of the Ada language [37], whereas the English research has mainly concentrated on non-compiler problems. VDM proof obligations are one of the major results of the latter research. A survey of the origins of the VDM notation and method can be found in the proceedings of the first VDM-Europe Symposium held in Brussels, 1987 ([33]).

Thus, the main reason for the existence of different VDM dialects is the different areas of application VDM can be used for. Unfortunately,
such a diversion does not stimulate industrial acceptance of VDM, and therefore in 1987 work started on the establishment of a ‘standard’ version of VDM by the British Standards Institute (BSI). The advantages of a widely accepted standard are obvious: the construction of tools is easier because such a standard language would be completely formally defined, and the acceptance of VDM in industry would be eased. BSI/VDM-SL was based on STC/VDMRL and DDC Meta-IV. The work was a joint effort of many experts on the various dialects of VDM-SL in Denmark, England, the Netherlands and Poland. In 1989 the standardization effort was accepted by ISO\textsuperscript{1} to become an ISO standard for VDM-DL. This work is still in progress and the final version of the standard is due end of 1992.

![Diagram of VDL dialects]

Figure 2.1: Main VDM Dialects

An overview of some the dialects is depicted in figure 2.1. A report that highlights some of the more important differences between the English and Danish viewpoints was presented in [207]. A report containing a short comparison of the main dialects was given in [176]. The differences between the dialects are concentrated in three points: the state concept used, the presence of a structuring mechanism and the definition of functions and operations, either implicit, explicit, or both. Most of these differences stem from the different application areas for which the dialects have been adapted. The following list gives a short introduction to relevant literature.

\textsuperscript{1}Under guidance of ISO SC22/WG19.
2.2. The Vienna Development Method

- **DDC Meta-IV**, or more commonly known as just plain Meta-IV, is probably the most well-known VDM dialect. It is defined in [35], [31], and [38]. It has been used for the definition of programming languages and the development of compilers as well as for the specification and design of large software systems.

- **SDRA Meta-IV** has been defined in [111]. The language is used for development of small programs or software systems, with the emphasis on proving the correctness of a development. The specification of operations can only be done implicitly, i.e. via pre- and postconditions.

- **STC VDM Reference Language (STC/VDMRL)** is defined in [197]. The language is intended for specification of software systems in general. Again the specification of operations can only be done implicitly, i.e. via pre- and postconditions.

- In 1986 an effort was started by the British Standards Institute (BSI) to establish a standard VDM specification language. BSI/VDM-SL is based on STC/VDMRL and DDC Meta-IV. The concrete and abstract syntax of BSI/VDM-SL are described in [5]. A first approach to its dynamic semantics was made by Arentoft and Larsen in [14]. The state concept chosen for BSI/VDM-SL is the same as the one chosen for STC/VDMRL ([14]). BSI/VDM-SL has an experimental structuring mechanism which is described in [24]. Both functions and operations can be specified either explicitly (functional or imperative style respectively) or implicitly (pre- and postconditions).

- The **Irish Meta IV** is described in [145]. The notation is strongly based on the Danish Meta IV dialect. It is fully functional and misses a state concept completely. The notation is accompanied by a method. A tutorial was presented at the VDM'91 conference [146]. The notation is not fixed. There is neither syntax nor formal semantics. The notation and method are carefully presented and set to work on a series of examples. The usage highlights the development of conceptual models and subsequent implementation of the models.

### 2.2.2 The VDM-SL notation

The VDM notation in its various forms and its method are widely documented. Dines Bjørner has established an annotated VDM bibliography in 1987 containing hundreds of items and presented an updated version in
April 1990 at the third VDM-Europe symposium [34]. Early tutorial papers are [143], [144] and [11], all three bundled in the 1987 proceedings [33]. More recent Cliff Jones updated his tutorial presentation of development in VDM in [115]. In [64] the current ISO standard in development for the VDM-SL is presented in detail. The VDM notation applied in this thesis is based on a preliminary form of the ISO VDM-SL notation described in [108]. For a detailed introduction to ISO-VDM-SL the reader is referred to e.g. the book of DAWES [64].

A VDM-SL specification consists of a series of definitions. A definition may be a type definition, a value definition, a function-, operation- or state definition. A specification is completely flat. There is no structure on the series of definitions.

- Type definitions are constructed from basic types like the natural numbers, integers, rational numbers and reals, booleans and characters. On top of the basic types new types can be constructed by application of type forming combinators like "set" giving finite sets, "+" giving finite sequences, "m" giving finite maps and \( \times \) giving cartesian products. Subtypes can be specified by adding an invariant to a type definition. Types cannot be loosely specified.

- Value specifications specify a certain value of a certain type and provides a name for the value.

- Function specifications specify mathematical functions over its domain to its co-domain. The body of a function is an expression. Functions can be total or partial. Functions can be specified either implicit or explicit. Functions can also be specified loosely. In section 4.6 the notion looseness is treated in detail providing the semantic interpretation of looseness and a series of examples.

- The last two constructions, operations and state definitions are closely linked together. An operation specifies a relation between its operands, its result and a set of state variables. The state specification models all globally manipulatable data of a specification. Operations can be specified implicitly or explicitly. Again looseness may be applied in operation specifications. The body of an explicit operation is a statement.

This thesis is certainly not meant to supply introductory course material for the VDM-SL. The reader has been given ample references to introductory material. The applied VDM-SL constructions that appear in MOSCA specifications in this thesis are kept simple and should not cause any problems to understand.
2.3 Process specification

2.3.1 Introduction

The VDM notation offers both data abstraction and control abstraction. The data abstraction is concerned with the description and manipulation of data values. The control abstraction offers constructions on micro and macro level, from simple conditional expressions and statements to functional and procedural abstractions. However, all constructions are limited to manipulate the control of a single thread. There are no process abstractions available in the notation.

The algebraic approach to process specification offers a means to describe the simultaneous behaviour of multiple control threads by virtue of behaviour expressions build from actions and operations on actions, like the prefix operator ‘○’ corresponding to the ‘≺’ relation defined earlier. A behaviour expression is not equivalent to a value expression with respect to its denotation. Value expressions\(^2\) denote numerical values, which result from a computation, by evaluating the value expression according to a fixed set of rules. E.g. the expression ‘4+5’ computes the value ‘9’ by evaluating it according to rules for the ‘+’ operator. Behaviour expressions like ‘p○X’ cannot be said to denote a numerical value, that can be computed according to rules for the ‘○’ operator.

One particular semantic approach takes the view that a behaviour expression denotes a state from which possible transitions, dictated by a set of transition rules, are possible. These transitions are part of a transition system. This approach builds on the theory of Structured Operational Semantics (SOS), introduced and studied deeply by PLOTKIN in [181].

Another approach associates behaviour expressions with nodes in a tree, in which a path along a branch from root to leaves exhibits the evolution of the behaviour expression. This approach highlights the denotational view on process semantics and originated from HENNESY in [95].

A complete different approach was investigated by GOLTZ in [85] and [84] and GLABBEEK EN VAANDRAGER in [81]. They investigated the connection between algebraic theories for concurrency, in particular CCS, and Petri-Net models.

This section is further organised as follows. In section 2.3.2 some general approaches to process algebras are briefly summerized. Both denotational and operational approaches are shortly introduced. Section 2.3.3 presents a

\(^2\)To distinguish data manipulating expressions from control expressions, in the ongoing discussion the former are denoted with the term value expressions and the latter by the notion behaviour expression.
short overview of CCS. It introduces the structured operational semantics of CCS and briefly discusses the basic notions of similarity of processes on which process refinement techniques and proof techniques are based.

2.3.2 Proce Algebras

In [94] Hennessy explores the semantic theory of processes. He defines a language to define processes, called EPL, which is strongly similar to the language from the CCS. The parallelism and concurrency are semantically treated as aspects of nondeterminism (e.g. a parallel process is equivalent to a nondeterministic one obtained by interleaving the actions of its constituent subprocesses). He defines three different semantic theories over EPL, two denotational and one operational one, and relates them.

The first denotational semantic theory starts from taking a particular abstract algebra as a denotational model. For finite terms of EPL simple $\Sigma$-algebras suffice, for terms including recursion he uses $\Sigma$-continuous-algebras. The algebraic setting provides a sound and complete proof system for the language, by choosing the denotational model to be initial with respect to a set of equations. The proof system is essentially the equational theory generated by the set of equations. These equations describe properties of the operators involved in the signature of the $\Sigma$-algebra. Hennessy builds on the pioneering work of Robin Milner, who was the first person to use an algebraic approach to the theory of communicating systems. As early as in 1978 Milner in [155] advocated the development of an algebra of activities which was subjected to a number of laws expressed as equations.

The second denotational semantic theory develops the notion of acceptance trees. For the finite terms of EPL finite acceptance trees are developed, and for the recursive terms the more general model of acceptance trees. In [95] this model was studied extensively. The finite acceptance trees $\text{fAT}$ associate with every process the action sequences in which it may partake, and with every action sequence some acceptance set. Elements of $\text{fAT}$ can be viewed as certain kind of rooted trees, with both the branches and nodes labelled by actions from $\text{Act}$, the set of all actions. In the infinite case the infinty of the branches is modelled by the introduction of two different kind of nodes, closed nodes and open nodes. Closed nodes represent deterministic terms, open nodes stand for least fixed point interpretations of recursive terms. The model is recently extended to cater for value passing processes in [97], [99] and [98].

The operational model builds on the structured operational semantics of Plotkin. It is a behavioural model in the sense that it enables the development of theories of behavioural equivalences. Processes may be divided
2.3. Process specification

Into equivalence classes induced by different notions of similarity. Milner has developed the two different notions, strong and weak observation equivalence in [157]. Strong equivalence says that two processes are equivalent if they mutually simulate the behaviour of each other at any time. Weak simulation is defined equally, without taking hidden or internal actions into account. Park [173] adapted these definitions to obtain the now very popular notions of strong and weak bisimilarity and almost every author on CCS alike process description notations copy him by arguing that these two notions hold (or do not hold) for their extensions or variants. Groote [87] has studied these notions of similarity in depth and I refer the reader to his thesis for further details. In [83] Van Glabbeek reports on a study of comparative concurrency semantics in which 11 different semantic theories related to the basic SOS process semantics are compared to each other with respect to their effect on process equivalence. In his thesis [82] he also investigates the effect of action refinement within the 11 theories for concurrency.

2.3.3 CCS

The constructions to model behaviour in Mosca resemble most closely the standard CCS and CSP operators. This choice results from a series of observations that are summarized below.

1. The theory of SOS gives a more simple operational and behavioural framework when compared for example to denotational models like acceptance trees. The theory handles data values, state and environments as well, without any difficulty (see e.g. the recent books of Hennesy [96] and Nielson & Nielson [167]).

2. The standard set of CCS operators prefix, choice, composition, relabelling and restriction are efficient enough to build general process behaviours. Most of the more esoteric operations found in CSP are expressible with the standard CCS operations.

3. The CCS extensions with time retain the simple SOS semantics of the pure CCS and are studied extensively in the literature (see section 2.5.3 for a full overview).

The Structured Operational Semantics for CCS is based on the notion of a labelled transition system. This semantic approach was first introduced by Plotkin in [181]. Recently an introductory text ([96]) on the subject was published, by Hennesy.

In CCS a process is denoted by the name agent. The basic CCS language is built from agent expressions, agent variables and agent constants. The
actions $Act$ range over an alphabet of labels $L$ build from the union of the set of names $A$ and co-names, $\overline{A}$ and the special action $\tau$. Let $f$ be a relabelling function such that $f(\overline{l}) = \overline{f(l)}$. Let $X$ be the set of agent variables, and $K$ the set of agent constants. The set of agent expressions $PE$ is then defined as the smallest set that includes $X$ and $K$ and contains the following expressions, given that $E$ and $E_i$ are already in $PE$:

- $a.E$ \quad $a \in Act$, a prefix
- $\sum_{i \in I} E_i$ \quad a summation ($I$ is an index set)
- $E_1 | E_2$ \quad a composition
- $E \setminus L$ \quad a restriction such that $L \subseteq L$
- $E[f]$ \quad a relabelling, with $f: L \rightarrow L$.

In CCS processes evolve by executing actions. The evolution of processes under execution of an action is modelled by a binary relation denoted by $\rightarrow$. The notation

$$P \xrightarrow{x} P'$$

reads: process $P$ may evolve to process $P'$ by executing action $x$, and forms the central idea behind a labelled transition system.

**Definition 2.1 (Lts)** A labelled transition system (Lts) is a triple

$$\langle PE, Act, \rightarrow \rangle,$$

where $PE$ is the set of process expressions, $Act$ is an arbitrary set of action names, and $\rightarrow$ denotes a relation in $PE \times Act \times P$. $\square$

The process expressions $PE$ are in general fixed by an abstract syntax. The transition relation $\rightarrow$ is in general fixed by a set of transition rules, which are defined over the abstract syntax forms for process-expressions. In general these rules appear as

$$\frac{P \xrightarrow{x} P'}{Q \xrightarrow{y} Q'}$$

which is a shorthand for

$$from \ P \xrightarrow{x} P' \ \text{infer} \ Q \xrightarrow{y} Q'$$

The part above the line is the **hypothesis** for the inference, the part below the line is the **conclusion**. The intuition underlying the inference rule is something like "if $P$ can proceed to $P'$, by executing $x$, then $Q$ may proceed as $Q'$ by executing action $y". The relation $\rightarrow$ is then the smallest relation inductively defined by the set of transition rules. For basic CCS, i.e. CCS without value passing the transition rules are given in figure 2.2.
2.3. Process specification

<table>
<thead>
<tr>
<th>Prefix</th>
<th>$a.E \xrightarrow{a} E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summation</td>
<td>$E_i \xrightarrow{a} E_i' \quad \sum i \in I E_i \xrightarrow{a} E_i'$</td>
</tr>
<tr>
<td>Compo1</td>
<td>$E \xrightarrow{a} E' \quad F \xrightarrow{a} F' \quad E \mid F \xrightarrow{a} E' \mid F'$</td>
</tr>
<tr>
<td>Compo2</td>
<td>$E \xrightarrow{a} E' \quad F \xrightarrow{a} F' \quad E \mid F \xrightarrow{a} E' \mid F'$</td>
</tr>
<tr>
<td>Restriction</td>
<td>$E \xrightarrow{a} E' \quad (a, \bar{a} \notin L)$</td>
</tr>
<tr>
<td>Relabelling</td>
<td>$E \xrightarrow{a} E' \quad E[m] \xrightarrow{m(a)} E'[m]$</td>
</tr>
<tr>
<td>Constant</td>
<td>$P \xrightarrow{a} P' \quad A \xrightarrow{a} P' \quad (A \Delta P)$</td>
</tr>
</tbody>
</table>

Figure 2.2: CCS Structured Operational Semantics

Besides the operational characterisation of the behaviour of CCS agents there is an equational characterisation that defines the behavioural properties of the operators of the calculus in the form of equations. An important fact of this characterisation is the expansion law. This law equates a specific form of behaviour expressions denoted by the term standard concurrent forms to an expanded form that readily expresses the immediate actions of the standard concurrent form. Repeated application of this law transforms compositions into choices, thus reducing the amount of concurrency in the expression by augmenting the amount of non-determinism in the expression. The definition of the expansion law is dependent of the internal action $\tau$ and introduces explicit $\tau$ actions into the resulting expression. I refer to section 3.3 of Milner’s recent book [157] for an explanation of the expansion law and examples of its application.

On the SOS semantical model several notions of similarity can be defined by means of an equivalence relation over a labelled transition system $\langle \mathcal{P} \mathcal{E}, \text{Act}, \rightarrow \rangle$. In [82] van Glabbeek studies 11 semantic models for concurrency that are defined by action relations over the domain of a restricted set of processes: finitely branching sequential processes, defined over the prefix operator and the choice operator, having in each state only finitely many possible ways to proceed. The weakest equivalence semantics, that is the semantics making the most identifications, is trace semantics, as pre-
presented in Hoare [103]. In trace semantics only partial traces are employed.

**Definition 2.2 (trace equivalence)** Let \( \langle \mathcal{PE}, \text{Act}, \rightarrow \rangle \) be a labelled transition system. \( \sigma \in \text{Act}^* \) is a trace of a process \( p \in \mathcal{PE} \) if there is a process \( q \in \mathcal{PE} \) such that \( p \xrightarrow{\sigma} q \). Let \( T(p) \) denote the set of all possible traces of \( p \). Two processes are trace equivalent if \( T(p) = T(q) \). In trace semantics processes are identified if and only of they are trace equivalent.

The strongest equivalence semantics is **bisimulation semantics**, as presented in Milner [157]. Bisimulation semantics is the standard semantics for the CCS language. Bisimulation equivalence can be further refined into (i) strong bisimulation equivalence and (ii) weak bisimulation equivalence.

The notion **strong equivalence** or strong bisimulation is characterised as follows. Given two agents \( P \) and \( Q \), then \( P \) is strongly equivalent to \( Q \) if for every action \( a \in \text{Act} \) every \( a \) derivative of \( P \) is equivalent to some \( a \) derivative of \( Q \) and conversely.

Adapting Milner's notational style ([157]), the definition of the equivalences may be stated as follows.

**Definition 2.3 (strong bisimulation)** Let \( \mathcal{PE} \) be the domain of behaviour expressions. Let \( P, Q \) range over \( \mathcal{PE} \). A binary relation \( S \subseteq (\mathcal{PE} \times \mathcal{PE}) \) is a strong bisimulation, if

\[
(P, Q) \in S \Rightarrow \\
\forall a \in \text{Act}.
\]

\[
P \xrightarrow{a} P' \Rightarrow \exists Q' \in \mathcal{PE} \cdot Q \xrightarrow{a} Q' \wedge (P', Q') \in S
\]

\[
Q \xrightarrow{a} Q' \Rightarrow \exists P' \in \mathcal{PE} \cdot P \xrightarrow{a} P' \wedge (P', Q') \in S
\]

**Definition 2.4 (strong equivalence)** Let \( \mathcal{PE} \) be the domain of behaviour expressions. Let \( P, Q \) range over \( \mathcal{PE} \). \( P \) and \( Q \) are **strongly equivalent**, or **strongly bisimilar**, written \( P \sim Q \), if there is a strong bisimulation \( S \) such that \( (P, Q) \in S \).

Milner shows that \( \sim \) is the largest strong bisimulation and that \( \sim \) is an equivalence relation ([157]). Weak similarity is similarly defined, but starting from a restricted transition relation \( \Rightarrow \).

**Definition 2.5 (weak transition relation)**. Let \( E \in \mathcal{PE}, a_i \in \text{Act} \). Let \( a = a_1 a_2 \ldots a_n \), then \( E \xrightarrow{a} E' \) if

\[
E(\xrightarrow{\tau})^* \xrightarrow{a_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow{a_n} (\xrightarrow{\tau})^* E'
\]
2.4. Time

Let $as \in Act^*$, then $[as]$ denotes the sequence of actions equal to $as$ but with all $\tau$ actions deleted. Now the notion weak bisimulation is defined as:

**Definition 2.6 (weak bisimulation)** Let $\mathcal{PE}$ be the domain of behaviour expressions. Let $P, Q$ range over $\mathcal{PE}$. A binary relation $S \subseteq (\mathcal{PE} \times \mathcal{PE})$ is a weak bisimulation, if

$$ (P, Q) \in S \Rightarrow $$

$$ \forall a \in Act. $$

$$ P \xrightarrow{a} P' \Rightarrow \exists Q' \in \mathcal{PE} : Q \xrightarrow{\delta} Q' \land (P', Q') \in S $$

$$ Q \xrightarrow{a} Q' \Rightarrow \exists P' \in \mathcal{PE} : P \xrightarrow{\delta} P' \land (P', Q') \in S $$

\[ \square \]

**Definition 2.7 (weak equivalence)** Let $\mathcal{PE}$ be the domain of behaviour expressions. Let $P, Q$ range over $\mathcal{PE}$. $P$ and $Q$ are weak equivalent, or weakly bisimilar, written $P \sim Q$, if there is a weak bisimulation $S$ such that $(P, Q) \in S$.

This notion is also called observation equivalence.

2.4 Time

The characteristics of the notion of time are certainly not agreed upon by the scientific community. A list that addresses only a few of the debated issues of time could run as

- is time discrete or continuous?
- is time unbounded?
- if time is continuous, is it dense and if so is it complete, that is can we model time by the rational or real numbers?
- is time branching or linear?
- etc.

The first section addresses the notion of time in the context of the software development activities. Semantic models of time are often strongly associated with logic. Manna and Pnueli have investigated a modal logic, often referred to as *temporal logic* [148]. Others have developed time logics based on first-order logics.
To study the various mechanisms used to give time a meaning within formal specifications is, to say at least, a confusing activity. With the strong expansion in numbers of distributed and real-time applications and the still growing demands on verified properties of these systems, formal theories dedicated to the description of time-dependent behaviour have become highly popular. Most theories, however, are logic theories and build either on a first order approach or a modal approach.\(^3\)

The first order approach goes back to the beginning of the century. RUSSEL [195] included time in his definition of change as the difference with respect to truth or falsehood, between propositions. Almost 60 years later QUINE [189] formulated logic expressions like

\[\neg(\exists x)(x \text{ is a time } \land P \text{ happens at } x)\]

for a notion like "*P will never happen*. This approach removes time from the domain of logic, by treating temporal elements of a sentence the same as non-temporal elements.

In contrast to this approach FINDLAY proposed in 1941 to give temporal elements a logical status. PRIOR then constructed in the early fifties his tense logic [187], on which all further development of temporal (modal) logics were based. He introduced the notions of $Fp$ for "*It will be the case that p is true*" and $Gp$ for "*It will always be the case that p is true*", $Pp$ for "*It has been the case that p was true*" and $Hp$ for "*It has always been the case that p is true*". He proved the identifications of $Gp$ with $\neg F \neg p$ and $Hp$ with $\neg P \neg p$. With the modal approach to temporal logic came the question about the topology of time: "Is time linear, branching, or even parallel, or circular?"

The two alternatives came forth from philosophical background, and were both to be incorporated into the formal reasoning of reactive systems. They are not equivalent in interpretation of relations between existence and time. An example in GALTON presents the next two sentences of the tense approach:

\[
\begin{align*}
F(\exists x)P(x) & \quad \text{it will be the case that there is something for which P holds} \\
(\exists x)FP(x) & \quad \text{there is something of which it will be the case that P will hold}
\end{align*}
\]

which clearly have different interpretations. In first order logic, with quantified time, the two formulas come out as something like

\[(\exists t)[\text{later}(t, \text{now}) \land (\exists x)f(t, x)]\]

\(^3\)see e.g the book from GALTON [77] for details on how these two approaches came into existence.
2.4. Time

and

$$(\exists x)(\exists t)[later(t, now) \land f(x, t)]$$

which are clearly equivalent. On the other hand not everything that can be expressed in the first-order approach is expressible in the modal approach.

The alliance between the two approaches (see van Benthem in [27]) comes about when the first-order approach is used to provide a model theory for the modal approach, i.e. the expressions within the modal approach are given a semantics in terms of first-order expressions. Later other modal theories for the modal approach were proposed, based on the specific entities of the application of modal logic within computer science.

The temporal logics are used extensively within the area of software engineering, both for specification and verification of sequential programs. The setting of the modal logic within computer science was first systematized by Pnueli in 1977 ([182]), starting from the work of Burstall’s assertion method ([48]). He introduced a temporal logic with future-tense operators like the $\Box$ operator, meaning always and it’s dual $\Diamond$, meaning sometime. The semantical models in the context of program verification take a discrete time model as starting point, corresponding to the discrete structure of the execution-sequence of computer programs. This lead Pnueli to introduce an additional temporal operator $\bigcirc$, meaning next, such that $\bigcirc P$ holds at a given time if $P$ holds at the next time instant. The work of Pnueli inspired many other scientists to develop variants and extensions of his original logic. The influence of the topology was studied in detail. Topics like fairness, safety, liveness were incorporated into the theories (see e.g. [42], [133], [148]).

The semantic model for Pnueli’s modal approach is usually based on sequences of states occurring in computations. The following description is taken from Pnueli’s survey on the applicability of temporal logic to the specification and verification of reactive systems ([183]). Let $\sigma: s_1, s_2, \ldots$ be a sequence of states occurring in a computation $s_0 \to s_1, \to \ldots$ of a given program. Each state $s_i$ assigns values to a set of variables $V$. The values assigned to the variables are constrained by the definition of the computation. Let $\text{len}s$ denote the length of a state sequence. A state formula is any well-formed first-order formula constructed over the variables of $V$. A temporal formula is a formula constructed from state formulas to which some of the following temporal operators are applied:

- $\bigcirc$ - Strong Next
- $U$ - Strong Until
- $\oplus$ - Strong Previous
- $S$ - Strong Since
Let \( p \) be a temporal formula and \( \sigma \) be a sequence of states. The meaning of the temporal operators is then defined inductively over the notion of \( p \) holding at position \( j < \text{len} \sigma \) of the sequence \( \sigma \), denoted by \( (\sigma, j) \models p \).

A innovation due to Moszkowski ([162]) was to base the semantics on intervals of time. His motivation was to facilitate the reasoning about finite chunks of program behaviour, as distinct from the entire execution sequences from Pnueli. He devised an Interval Temporal Logic, ITL, that was again extended and varied upon by numerous other workers in the field. A recent extension to the ITL is the duration calculus of Zhou et al. ([54]). The study of intervals was naturally taken up within the framework of first-order logics as well ([110]).

These developments resulted into four different kind of logic classes, each with its own group of practitioners. Like in the philosophical circles, certain members of each group argued that the other logics were inferior to their logic with respect to the applicability within the real-time scene. Jahani and Mok [110] developed e.g. a first-order logic called RTL, Real-Time Logic, which they claimed to be more useful than any temporal (modal) logic. Similar claims were raised by Auernheimer and Kemmerer [16] in their work on RT-ASLAN, an extension of ASLAN, a specification language based on the first-order predicate calculus and a state machine approach, Alagar and Ramanathan [8] in their functional specification approach to real-time systems, Caspi and Halbwachs [51] in their work related to functional description of real-time systems. Yet numerous workers stick to variants of modal logics as a tool for formal reasoning, foremost of all Amir Pnueli himself.

The temporal approach was not designed to be applied for the specification of real-time systems. Recently various extensions and adaptations have been made of the temporal framework to deal with real-time (e.g. [193], [184], [169] and [10]).

### 2.5 Combination notations

#### 2.5.1 Value and Process combinations

The combination of value semantics in general and process semantics has been studied broadly. Value passing between processes has been described extensively. Hoare ([103]) offers in CSP synchronous communication through messages. Milner treats synchronizations with value passing as a straight generalisation of synchronization without value passing by giving it a semantics through translation of value passing actions to pure synchronization actions ([157]). Like Hoare he assumes very powerful data models, like sets
and sequences without stating any formal properties of his assumptions. Hennessy ([98], [97], [99]) gives a full denotational semantic model for a process description language based on the basic CCS operators with value passing. The capabilities of the value passing is constrained to the natural numbers, however. Hennessy's semantic model is denoted by the term acceptance tree.

2.5.1.1 Algebraic data and process description

The combination of an algebraic data description approach and a process algebra based on CCS and CSP resulted in the technique LOTOS.

The main results of the ESPRIT effort that developed LOTOS is collected in [218]. The specification language LOTOS Language Of Temporal Ordering Specification was designed to enable formal description of OSI architectural concepts such as services, protocols service access points, etcetera, and became an ISO standard in 1988 [1]. Contrary to the name suggest, LOTOS is not related to temporal logic. LOTOS is based on CCS, with little influences of CSP. CCS is used as semantical basis for the process part of the language [219]. LOTOS is a rich notation. Like CCS behaviour expressions in LOTOS are build from the standard operators like action prefix, choice and composition. Hiding ports and restricting ports from synchronization are two different phenomena in LOTOS, the first realized by a hide operator, the second by a special form of the composition operator. Next to these basic operators LOTOS contains operators that manipulate processes like process instantiation, sequential composition and process disabling. Process instantiation offers actual gate names to the formal gates of a process definition, thus realizing relabelling of ports in a more scope-directed fashion whereas the CCS relabelling operator offers an expression-based functionality. LOTOS offers the exit expression that is only capable of performing a single internal action \( \delta \), after which it transforms into the dead process \( \text{stop} \). The exit construction is used to model sequential composition of processes. The behaviour expression

\[ P1 \gg P2 \]

denotes a process that behaves like \( P1 \) and if \( P1 \) terminates successfully, that is, not because of premature deadlock, transforms into \( P2 \). In the application area for which LOTOS was developed it is often the case that actions can be disrupted or connections be broken. To cater for these requirements LOTOS was equipped with an "application generated" operator, the disabling operator. The behaviour expression

\[ Q1[> Q2 \]
behaves likes Q1, but Q1 can be disabled by Q2 in the following sense. If Q2 ever performs an action and becomes Q'2, the behaviour continues with Q'2. If Q1 terminates successfully then Q2 disappears.

The requirements for abstractness favoured the choice of an abstract data type definition technique as value description device. Here ACT-ONE was chosen [68] as a starting point. ACT-ONE is an algebraic specification method to write unparameterized as well as parameterized abstract data type specifications. It includes the use of a library of predefined data types; extensions and combinations of already existing specifications; parameterization and actualization of parameterized specifications and renaming of specifications. The most basic form of a data type specification consists of a signature and a possible empty list of equations.

LOTOS is an executable notation ([69], [150], [214]). Recently experiments have been carried out to incorporate a notion of time in LOTOS [188]. Like VDM, the LOTOS notation is being incorporated into a design methodology, called Lotosphere again through an ESPRIT project⁴.

2.5.1.2 Model-oriented data and algebraic process description

The combination of model-oriented data specification by means of VDM-SL or one of its predecessor forms, and a process oriented notation has been studied as well. In [72] FOLKJÄR and BJÖRNER report on a research centered around the combination of VDM and CSP. An example of the application of such a combination can be found in the work of DAWIDS and LØVENGREEN ([65]), where it is set to work on a rigorous development of a distributed calendar service.

Another early report was given by FANTECCI in 1984 ([70]) on the combination of an earlier form of VDM-SL, called Meta-IV ([35]) and CCS. However, his presentation includes only a very intuitive sketch of the meaning of such a combination. In [66] DENVIR reports on a similar approach, he being even more intuitive. Both approaches can be viewed as “wishful thinking”, without giving a serious attempt to describe such a combination in detail.

MAC AN AIRCHINNIGH ([145], [146]) proposes a specification language again based on Meta-IV enhanced with process describing capabilities similar to CSP, although, again without stating any formal semantics. In contrast to the two aforementioned studies, his presentation is very thorough, based on general accepted mathematical properties of the involved constructions.

JONES has investigated another line of specifying and design of parallel

⁴ESPRIT II Lotosphere project, nr. 2304.
programs. In his thesis ([112]), and followed by later work in [113] and [114] he studied the notion of interference related to compositional development of concurrent shared variable systems. He developed the notions of rely and guarantee conditions for implicit operation specification. This line of research has been followed up by Stølen in [200], and [201], by extending the original rely-guarantee method of Jones to include the specification of systems whose correctness depends upon synchronization.

RAISE\(^5\) *Rigorous Approach to Industrial Software Engineering* is a collection consisting of a specification language called RSL\(^6\), a development method, and a set of supporting tools. The RAISE Method is, just like VDM, based on the notion of stepwise refinement. Its specification language RSL contains notions to express data-abstraction through model-oriented and property-oriented facilities. Control abstraction for parallel activities is based on the process concept of CSP extended with the facility to specify processes implicitly through trace and failure assertions. RSL does not handle the notion of time. The RSL language builds on the DDC Meta IV dialect. The RAISE ([166]) specification language RSL has been given a complete formal semantics in a denotational style, recorded in a restrictive subset of RSL ([6]). It includes a class semantics for the RSL structuring mechanism and a denotational semantics that provides models for the RSL specifications. The domain of semantic processes required to describe the RSL language consists of potentially infinite trees ([154]) which can have as their branches entities which, when provided with an input, continue with a process or which can provide an output and continue with a process. There are to my knowledge no detailed presentations of this model in the scientific literature. All RSL semantic publications are propriety material of Computer Resources International (DK).

### 2.5.2 Value and Time combinations

Studies at combinations of specification notations in the style of VDM-SL and time are less abundant than the combinations of data and process combinations. BROY ([45]) describes a functional stream language, similar to the applicative part of Meta-IV, with time extensions. MIDDELBURG ([152]) describes a time extension of VDM-SL based on temporal logics. His language, VVSL, is designed to obtain a language for VDM-alike specifications with additional constructions to specify non-atomic operations. This is accomplished by adding an *inter-condition* to the pre- and post-conditions of

---

\(^5\) Developed under the Esprit program of the CEC in the field of Software Technology proj. no. 315

\(^6\) RSL stands for Raise Specification Language
implicit specified operations. These inter-conditions are formulae of a temporal logic language. The whole language contains a strong modularisation concept and is given a formal semantics based on a many-sorted infinitary logic of partial functions, $MPL_\omega$, ([129]). which is an extension of first-order logic, and a general algebraic model of modules called Description Algebra ([117]). Recently Jones has continued his research on interference ([116]). He studies a collection of temporal operators aimed at natural reasoning about programs. These operators apply to predicates of two states and a style of specification is proposed which clearly separates transitions of a component from those of its environment. He introduces the notion of resumptions as a semantic model for the logic.

2.5.3 Process and Time combinations

The combination of time and process specification is found in many different settings. Most of the formal devices that were in use for the specification of sequential systems have been equipped with a notion of time, as such enlarging the applicability of these devices to the field of real-time specifications. Without any claim on completeness I mention

- time extensions to Petri-Nets (e.g. [29], [53], [78], [139], [151], and [165]),
- State-Machines (e.g. [76], [109], [142], [127], [225], and [226]),
- Event/Action systems (e.g. [8], [110], and [30]),
- general approaches on the formal specification of timed processes ([22], [20], [67], [118], [131], and [54]),
- combinations based on CSP ([65], [168], [192], and [194]),
- combinations based on CCS ([222], [221], [220], [104], [56], [188], [89], [158], [130], [20], [61]).

The most relevant combination of time and process description with respect of the development of MOSCA are those concerning CCS and time. The ear-liest description can be traced back to 1985, where Milne ([153]) describes CIRCAL, based on a variant of CCS called the dot calculus. The treatment of time is more or less equivalent to the first-order logic approach. It is not treated as a semantical issue, but modelled through ticking agents (clocks) that are described with timeless CCS alike agents. A multitude of approaches appear between 1988 and 1991, based on the extension of the prefix construction. They offer various forms of delay constructions, idling,
time-out operations, timed events, interval timing etc. They all give their extensions a structural operational semantic basis. Some treat time as a continuum, taking the real numbers as instants in time, others take the natural numbers as basis. I have studied eight different approaches, (still only a selection of a greater total), mostly chosen on pragmatic reasons, the availability of material. The selection includes the work of Baeten and Bergstra, Chen et al., Daniels, Hansson, Moller et al., Krishnan, Quezada et al., and Wang et al. A brief summary of a comparison of these approaches is presented in figures 2.3 and 2.4. A brief summary of these approaches follows next. Let $\Lambda$ denote the set of action labels excluding the internal action $\tau$, let $Bexpr$ denote the syntactic class of behaviour expressions.

1. Wang’s Model ([222], [221], [220], [104]) treats time as a continuum. His time domain is $\mathbb{R}^+$, the positive reals, including 0. Time is registered in time variables which are added to actions in prefix constructions. The actions itself are timeless. Progression of time results from a special action, the idle action. A timed action prefix has the following form

$$a@t.P(t)$$

where the dot "." denotes the usual prefix operator, $a$ is an action and $t$ a time variable. The passing of time from the moment that a prefix becomes ready to be taken, i.e. the agent in which the expression occurs starts waiting on a synchronization by the environment on the action $a$, until the actual moment that the synchronization is performed is registered in time variable $t$ and subsequently substituted in each applied occurrence of the variable in the behaviour expression $P$. Initially $t$ is zero. The passing of time is incorporated in the semantics by extension of the transition relation in the following sense:

$$a@t.P \xrightarrow{\delta} P[\delta + t/t]$$

where $\delta \in \mathbb{R}$ and $P[\delta + t/t]$ denotes the substitution of $\delta + t$ in each occurrence of the time variable $t$ in $P$. The transition relation thus has the following signature:

$$\rightarrow : (Bexpr \times \Lambda \cup \mathbb{R} \times Bexpr).$$

Idle actions can be used to model inaction. The agent that waits for 5 seconds and then behaves like $P$ is $\epsilon(5).P$. Each idle action represents a whole continuum of transitions.

$$\forall \delta \leq d \cdot \epsilon(d) \xrightarrow{\delta} \epsilon(d - \delta).P$$
<table>
<thead>
<tr>
<th>Issue</th>
<th>WANG+</th>
<th>QUEMADA+</th>
<th>HANSSON+</th>
<th>MOLLER+</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic Properties</td>
<td>timed action prefix</td>
<td>timed events, time choice</td>
<td>time-out operator &gt;</td>
<td>unbounded delay op. δ, wait operator (t).P</td>
</tr>
<tr>
<td>Time domain</td>
<td>( \mathbb{R} )</td>
<td>( \mathbb{N} )</td>
<td>( \mathbb{N} )</td>
<td>( \mathbb{N} )</td>
</tr>
<tr>
<td>Label domain</td>
<td>( \lambda \cup \mathbb{T} )</td>
<td>( \lambda \times \mathbb{T} )</td>
<td>?</td>
<td>( \lambda, \rightarrow \mathbb{T}, \approx )</td>
</tr>
<tr>
<td>Time progression</td>
<td>delay &amp; idling, ( \varepsilon(d) )</td>
<td>Idling through time comp. of actions</td>
<td>delay &amp; idling</td>
<td>delay &amp; idling</td>
</tr>
<tr>
<td>Time determinism</td>
<td>implicit non-determinism</td>
<td>explicit non-determinism through time-choice</td>
<td>explicit by ticking</td>
<td>explicit by ticking</td>
</tr>
<tr>
<td>Clock present within semantic model</td>
<td>N.a.</td>
<td>implicit global clock, updated by ( \text{Old} )</td>
<td>implicit global clock</td>
<td>implicit global clock</td>
</tr>
<tr>
<td>Equivalences</td>
<td>( \approx, \approx )</td>
<td>( \approx, \approx )</td>
<td>( \approx )</td>
<td>( \approx )</td>
</tr>
<tr>
<td>Equational Theory</td>
<td>Present</td>
<td>N.a.</td>
<td>N.a.</td>
<td>Present</td>
</tr>
<tr>
<td>Related Logic</td>
<td>For regular calculus only</td>
<td>N.a.</td>
<td>In preparation</td>
<td>N.a.</td>
</tr>
</tbody>
</table>

Figure 2.3: Comparison of Timed CCS approaches – 1
<table>
<thead>
<tr>
<th>Issue</th>
<th>CHEN+</th>
<th>KRISHNAN</th>
<th>BAETEN+</th>
<th>DANIELS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic Properties</td>
<td>idling through action interval, time variables</td>
<td>delays and explicit interval timing constraints</td>
<td>Integration, timestamps</td>
<td>time-intervals on each action</td>
</tr>
<tr>
<td>Time domain</td>
<td>( \mathbb{R} )</td>
<td>( \mathbb{N} )</td>
<td>( \mathbb{R} )</td>
<td>( \mathbb{R} )</td>
</tr>
<tr>
<td>Label domain</td>
<td>( \lambda \cup \mathbb{T} )</td>
<td>( \lambda \cup {\tau, \delta} )</td>
<td>( \lambda \times \mathbb{T} )</td>
<td>( \lambda \cup \mathbb{T} )</td>
</tr>
<tr>
<td>Time progression</td>
<td>delay &amp; idling</td>
<td>clock ticks</td>
<td>projective limit model</td>
<td>(semantic) idling</td>
</tr>
<tr>
<td>Time determinism</td>
<td>implicit nondeterminism</td>
<td>explicit by ticking</td>
<td>integral mechanism</td>
<td>implicit nondeterminism</td>
</tr>
<tr>
<td>Clock present within semantic model</td>
<td>N.a.</td>
<td>implicit global clock</td>
<td>N.a.</td>
<td>N.a.</td>
</tr>
<tr>
<td>Equivalences</td>
<td>( \sim, \approx )</td>
<td>simulations induced by timing constraints</td>
<td>several</td>
<td>( \sim, \approx ) and timed-weak bisimulation</td>
</tr>
<tr>
<td>Equational Theory</td>
<td>N.a.</td>
<td>N.a.</td>
<td>various sorts</td>
<td>Present</td>
</tr>
</tbody>
</table>

Figure 2.4: Comparison of Timed CCS approaches – 2
This renders the transition relation wildly infinite and wildly nonde-
terministic. To keep the model realistic with respect to time percep-
tion the semantics reflect some important fundamental properties:

- **maximal progress** Whenever an agent can perform a $\tau$ action, it
  shall never wait,

- **time determinism** Whenever $P \xrightarrow{\delta_1} P'$ and $P \xrightarrow{\delta_2} P''$ then $P' \equiv P''$, i.e. when time goes, if an agent is idling, then it cannot
  reach different states.

- **time continuity** $P \xrightarrow{\delta_1 + \delta_2} P'$ iff there exists a $P_1$ such that $P \xrightarrow{\delta_1} P_1$ and $P_1 \xrightarrow{\delta_2} P'$. So if an agent proceeds from one instant
to another it must reach all the intermediate instants between

- **time persistency** If $P \xrightarrow{\delta} P'$ and $P \xrightarrow{a} Q$ then $P' \xrightarrow{a} Q$. Thus
  by idling an agent shall not loose the ability of performing an
  action that it is able to perform originally.

Wang defines an agent $NIL$, which persists in time by performing
idle actions. The calculus is denoted by TCCS, has been given a
complete structural operational semantics, includes both strong and
weak notions of bisimilarity and has an equational caracterization
in terms of the bisimilarity equivalences. The theory lacks a formal
reasoning device. However, in [104] a regular subcalculus of TCCS is
defined which has been given a timed modal logic interpretation.

2. Chen c.s. define an extension of Wang's model in [56] named timed
CCS. They also apply a dense time domain. The model contains the
same basic approach as Wang's TCCS, but is extended to contain
interval specifications. E.g. the behaviour expression

$$a(t) \mid_{1}^{6}. P(t)$$

denotes an agent which can perform the action $a$ at any time $t$ such
that $1 \leq t \leq 6$. The general form

$$a(t) \mid_{l}^{u}. P(t)$$

represents an agent which can perform action $a$ at some time between
$l$ and $u$, in doing so it will evolve to $P[d/t]$, where $d$ it the time at
which action $a$ is taken. The time expressions $l$ and $u$ represent
relative times with respect to the moment the prefix is ready to be
taken. Although the notation is given a full SOS based on the same
principles as Wang's notation, it is not clear what the meaning is of such prefix expression whenever the upper limit of time cannot be satisfied. E.g. the expression

\[ a(t) |^5_3 . P(t) | a(u) |^{10}_6 . Q(u) \]

states that in the left operand of the composition operator \(|\) the action \(a\) should be taken after 3 but at most after 5 time units. The action in the right operand can be taken after 6 units of time, so in the whole composition no synchronization on \(a\) can ever occur. The semantic approach in this situation is such that time cannot proceed to a value exceeding 5 idling time units, and the whole system is inevitable blocked.

The upper limit is permitted to be open, i.e. to become \(\infty\). Expression with open upper limits can be expressed directly within Wang's calculus. The expression

\[ a(t) |^\infty_1 . P(t) \]

is equivalent to the TCCS expression

\[ idle(l).a@. P(t) \]

Expressions with closed upper limit cannot be expressed directly within Wang's notation, although the effect is achieved by the application of a choice expression. The general form \(a(t) |^u_1 . P(t)\) transforms into

\[ idle(l).a@. P(t) + idle(h). \bot \]

where \(\bot\) resembles the divergent process.

3. QUEMADA ET AL. define a timed calculus that is fitted onto LOTOS, called TIC in [188]. Here the time domain is discrete. Actions are extended to hold a delay. Subsequently, the domain for the transition labels is \(\Lambda \times T\), where \(T = \mathbb{N}\). Here the meaning of a behaviour expression like

\[ at.P \]

is the following. Action \(a\) shall happen at time instant \(t\) relative to the time instant of the previous event. This approach presumes an implicit global clock. Again intervals can be specified. The expression

\[ a[1..5].P \]
called a time choice is equivalent to


Intervals may have an open upper limit. The notation is given a structured operational semantics and the usual bisimulation equivalences are defined. There is no equational characterization for the notation and no logic.

4. **Hansson and Jonsson** in [89] define a calculus with both time and probabilities. Their notation TPCCS is built on a discrete time domain, assumes a global clock, has delay and idle constructions and contains a time-out operator \( \triangleright \). Intuitively \( (e_1 \triangleright e_2).P \) denotes an agent that after \( i \) time steps becomes \( e_2.P \), unless action \( e_1 \) is taken prior to that. The action \( e_2 \) can be taken at any time during the time-out period. So, the effect of the operator can be expressed in Wang’s notation as

\[ (idle(i).e_2@t.P(t) + (e_1@t.P(t) + e_2@u.P(u))) \]

The actions take no time. Again the principles of arbitrary waiting and minimal delay are present, just like in Wang’s calculus. The authors do not present a formal semantics, nor any other formal concept yet.

5. **Moller C.S.** in [158] again take the natural numbers as time domain, apply a bound delay coupled to actions just like Quemada, included a boundless delay specifier \( \delta \) corresponding to Wang’s model, and assume an implicit global ticking clock. The transition relation is split into two relations, one for the normal actions, and one for the delays.

6. **Krishnan** also employs a discrete time domain. In RTCCS ([130], real-time CCS) actions take unit time. Time, recorded by an implicit global clock, progresses by action ticks and delays. The language is similar to Moller’s approach. The most distinctive quality of the calculus are the explicit timing constraints, which are similar to those applied by Jahanian & Mok in their event/action specification system. Krishnan defines absolute and relative timing constraints. E.g.

\[ A[a, t1, t2] = \text{true} \]

signifies an absolute constraint that requires the action \( a \) to happen after \( t1 \) and before \( t2 \). \( A[a, t1, t2] = \text{false} \) requires that \( a \) should not
2.5. Combination notations

occur within the specified interval. The relative timing constraint

\[ R[a, b, t_1, t_2] = \text{true} \]

requires a \( b \) action to occur after \( t_1 \) time units, but within \( t_2 \) units of time after the action \( a \) was taken. Through these timing constraints the available non-determinism within behaviour expressions is restricted. RTCCS without the timing constraints is given a structured operational semantics. The notions of similarity are induced by this semantics and the explicit timing constraints.

7. BAETEN and BERGSTRA in [20] develop an algebraic theory based on their original ACP. They include a dense time domain, apply relative delay specifications added to the actions similar to Quemada and Moller, extend the time-choice of Quemada to the real domain by the definition of time-integration.

8. DANIELS in [61] proposes CCSiT, yet another CCS derivative with time-intervals associated with actions that specify when actions are allowed to occur. The approach is closely related to the work of BAETEN+. He includes a dense time domain. An interval is associated to each action. For example

\[ \mu @ [0, 2] \]

denotes an action that should be taken within 2 seconds. The interval may include or exclude the end-points. Actions are atomic and take not time to execute. Like in WANG's approach time advances when no processes execute any actions. Again idling is synchronous, that is all parallel processes idle the same amount at the same time. The idle construction is in contrast to WANG purely semantical. Process expressions are formed out of a terse syntax:

\[ \text{NIL} \mid \mu @ i. P \mid P + Q \mid P || Q \mid P \setminus l \]

as such excluding recursive terms, with \( i \) an interval specification and \( l \) a restriction set. The operational semantics is given in terms of a lts. The passing of time is described by shifting the intervals associated to actions. For instance

\[ \mu @ [l, h]. P \xrightarrow{t} t \mu @ [l - t, h - t]. P \]

may be a transition step. Time may pass upper limits of intervals only for external actions. Internal actions are not allowed to be delayed longer that the upper limit of the associated interval.

DANIELS provides three different equivalence characterizations: strong, timed weak and weak bisimulation.
2.5.4 Value, Process and Time combinations

Combinations of value, process and time specification are also abundant. Many of the already existing process specification techniques included value specification as well. E.g. state-machine based specifications like HAREL's state-charts ([90], [91]) have been extended to contain time specification as well in [127]. PNUELI's approach to the specification of concurrent systems through combinations of temporal logic and various forms of state transition systems\(^7\) has been extended to cater for real-time systems in the timed transition systems ([102], [101], [147]).

Addressing all three phenomena together within the framework of model-oriented value specification is a task that is sparingly undertaken. One approach has been reported by DAWIDS and LØVENGREEN in [65]. They extend Meta-IV with process description capabilities based on CSP primitives and use temporal logic to state properties of their specifications.

As far as I have seen there have been no specific detailed work within the framework of model-oriented value specification on the combination of all three different phenomena within one single notation, which in itself provided me with a strong rationale for the current work on MOSCA.

---

\(^7\) which have a complete other semantic base as labelled transition systems, see e.g. [149].
3.1 Introduction

This chapter introduces the notation of MOSCA piece by piece, through a range of simple examples. The MOSCA notation is a merge of the mathematical concrete syntax of VDM-SL and stylistic forms of the basic CCS operators, enriched with specific forms to describe the interface of agents with the outer world and time dependent behaviour. The term agent is used to denote a process. It is taken from the CCS terminology and denotes a behaviour expression without any free variables. A complete definition of the mathematical concrete syntax of MOSCA is covered by appendix A. This syntax is a presentation syntax and certainly not applicable as tool syntax. A tool syntax for a subset of MOSCA can be found in [170]. The first reference to a member of a certain syntactic class of the mathematical concrete syntax is highlighted by postfixing the name of the class with the $P$S acronym, which stands for Presentation Syntax.

The chapter is composed as follows. In section 3.2 an overview is presented of the various kind of attributes an agent can obtain through definition variations. Section 3.3 summarizes the different forms of behaviour expressions in MOSCA.

An agent models a computing process that may encapsulate a private state. The MOSCA model for specifying shared memory systems is a unit. Units are presented, together with a primitive structuring mechanism and a discussion concerning matters of scope and visibility in MOSCA in section 3.4.

Section 3.5 highlights the constructions in MOSCA to incorporate time information into specifications. The main vehicles to do so are two extensions of the prefix construction, the timed prefix construction and the idle prefix construction.
Finally a section presenting a summery of MOSCA closes the chapter. Parts of the material in this chapter have been published earlier in [208] and [210].

### 3.2 Agent definition

Agents are defined by the $\text{AgentDefinition}^\text{PS}$ construction. Agent definitions can be given in various styles, ranging from simple agents without any means to handle values — through agents with value parts — to agents with local state and associated operations that act on the state of the agent. The capabilities of agents with respect to value manipulation is summarised in figure 3.1.

<table>
<thead>
<tr>
<th>capability</th>
<th>type</th>
<th>Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type 0</td>
<td>Type 1</td>
</tr>
<tr>
<td>Value Passing Actions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Value Part Specification</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>State Part Specification</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Value Manipulation</td>
<td>—</td>
<td>functions</td>
</tr>
</tbody>
</table>

**Figure 3.1: Agent / Value combinations**

Agent definitions can be simple or compound. A simple agent definition consists of an optional agent heading and port specification, one or more agent behaviour definitions and possibly a series of local definitions. The agent behaviour definitions all have (i) the same name, i.e. the name of the agent and (ii) share the same heading and port specification. The agent heading and port specification is stable during the lifetime of the agent. A compound agent, called a **unit**, also defines one agent, but now the agent’s heading and port specification may vary during it’s lifetime. The unit heading and port specification is refined by the series of agent behaviour definitions. Each agent behaviour definition may **extend** the port specification of the unit, without altering the original ports. Furthermore, each agent behaviour definition may **replace** the original value part specification. The state part specification in the unit heading is stable, visible during the whole lifetime of the agent.
3.2. Agent definition

An agent definition basically consists of three parts, an agent heading, a port specification and the behaviour of the agent. An AgentHeading\(^{PS}\) starts with an identifier. It is the name of the agent. The PortSpecification\(^{PS}\) of a agent definition specifies the interface of the agent with the surrounding system. It enumerates the ports of the agent. Each entry states the name of the port and its specific usage: either for synchronization, for reading or for writing data values.

The third part defines the behaviour of the agent. The behaviour is defined through AgentBehaviour\(^{PS}\) constructions. They consist of the agent name and the \(\triangleq\) symbol followed by the definition of the behaviour in the form of a BehaviourExpression\(^{PS}\).

The constructions to specify the behaviour of agents fall in the syntactical class denoted by BehaviourExpression\(^{PS}\). A behaviour expression is built from classic constructions like prefix and choice constructions and compound constructions like agent if and agent let constructions. In the following sections many constructions are introduced by example to describe the various aspects of agent definition.

3.2.1 Type 0 Agents

The general form for the most simple style agent definition is depicted in spec. 3.1. This style involves no value passing, either through actions, value parts or through the state of the agent, and is referred to as Type 0 agent. In the Type 0 style the heading is limited to the name of the agent.

\[
\begin{align*}
\text{Agent\_name} & \\
\text{ports syn portname}_1 & \\
\ldots & \\
\text{syn portname}_n & \\
\text{Agent\_name} & \triangleq \text{Agent-Behaviour-Expression}
\end{align*}
\]

\underline{Example 3.1} Suppose we want to model a simple semaphore device (specification 3.2), only capable of undertaking series of \(p\) and \(v\) actions. First we state the name of the agent, which is Semaphore, followed by the port specification. The port specification starts with the keyword ports, followed by a series of port specifications. The specification \text{syn p} specifies a port dedicated to synchronization purposes only.

A port specification starts with a keyword that states the capability of the port: \text{in} for input ports, \text{out} for output ports and \text{syn} for synchronization.
ports. We call \( p \) and \( v \) the labels of the port. Labels with an overline are associated with output ports, labels without an overline are associated with input ports or synchronization ports.

The behaviour of the semaphore is given by \( p \odot v \odot \text{Semaphore} \). It defines the agent \( \text{Semaphore} \) to be able to perform an action \( p \) followed by an action \( v \) after which the agent becomes itself again. We have here an example of a cyclic agent definition.

Syntactically the behaviour consists of two prefix expressions \( p \odot (v \odot \text{Semaphore}) \) and \( v \odot \text{Semaphore} \). The prefix operator \( \odot \) is equivalent with the CCS prefix operator ‘.’. The action, represented syntactically by an \( \text{Action}^{ps} \) construction, specifies a communication through the designated port, the behaviour expression following the ‘\( \odot \)’ symbol specifies a replacement behaviour for the agent.

For Type 0 agents the action is restricted to a single label. The next class of agents, Type I agents, can handle parameterized actions.

### 3.2.2 Type I Agents

The ability of storage values within an agent can be modelled in two different ways, through a value part and through a full state definition. Both entities allow the storing of values within an agent. The main difference between the two approaches is the way the stored values can be manipulated. Agents equipped with value parts are referenced as Type I agents.

The definition of a Type I agent starts with an identifier to name the agent, followed by a type specification that states the type of the value part of the agent. A value part specification has the form of a \text{mosca} type expression. The value can be set through agent instantiation by means of an agent-service construction, which binds a value to the value part. The general form of a Type I agent is given in spec 3.3. The port specification for value passing ports specifies a direction specifier, a port name and a port value type. E.g.

\[ \text{in set: N} \]
3.2. Agent definition

\[
\begin{align*}
\text{Agent\_name} \ (\text{Valueparttype}) & \quad \text{3.3 Type I Agent} \\
\text{ports in inportname} & : \text{Type} \\
\text{out outportname} & : \text{Type} \\
\text{Agent\_name} \ (\text{valuepartpattern}\_1) & \triangleq \\
\text{Agent\_Behaviour\_Expression}\_1 \\
\vdots \\
\text{Agent\_name} \ (\text{valuepartpattern}\_n) & \triangleq \\
\text{Agent\_Behaviour\_Expression}\_n \\
\end{align*}
\]

Ports without a port value type specifier cannot pass any value, their existence is purely for synchronization purposes like in the semaphore example.

The third part of a Type I agent definition specifies again the behaviour of the agent. In general the behaviour is specified through a series of \(\text{Agent}\_\text{Behaviour}\_P^S\) constructions. Each construction defines the behaviour of the agent for a specific set of values of the value part of the agent. This set of values is fixed by a \(\text{Pattern}\_P^S\) construction. Through pattern matching all behaviour expressions are selected that are associated with matching value part patterns. This is a form of looseness within the agent definitions that results in non-deterministic behaviour. If a deterministic selection is desirable, care must be taken to let the set of patterns match all possible values for the value part type and to exclude patterns with overlapping associated value sets.

Example 3.2 MOSCA can be used to model a wide variety of concurrent systems. In small grain concurrent systems one even could model memory cells as independent entities. In the example is assumed that a memory cell can store integer values, and can be used through the normal read and write operations. We model the storage capability of a memory cell with a value part. The read operation is modelled with the output action \(\text{get}\), the write operation through the input action \(\text{set}\). Specification 3.4 gives

\[
\begin{align*}
\text{MCell} \ (Z) & \quad \text{3.4 Memory Cell} \\
\text{ports in set} & : Z \\
\text{out get} & : Z \\
\text{MCell} \ (\text{val}) & \triangleq \text{set}(x) \odot \text{MCell} \ (x) \oplus \text{get}(\text{val}) \odot \text{MCell} \ (\text{val})
\end{align*}
\]

the MOSCA specification.
The agent *MCell* has an integer value part, so the value patterns in the agent behaviour construction must match integer values. The actual pattern 'val' matches all possible integer values, thus bundling the behaviour into one agent behaviour construction. The value pattern in the agent behaviour construction may introduce in general a collection of variable names. The scope of these names reaches to the end of the associated behaviour expression. The behaviour construction of the *MCell* agent applies a Choice\(^{PS}\) expression. This expression is formed with the choice operator \(\oplus\). The choice expression is equivalent to the binary form of the CCS summation construction. Input- and output-actions are used to pass values from one agent to another. The value parameters of actions are patterns for input actions and expressions for output actions. Notice that

- the pattern in an input action has a defining nature. The names in these patterns define a new scope region, that starts with the occurrence of the name and reaches to the end of the prefix expression. In the prefix construction

\[
\text{in}(x) \odot A
\]

\(x\) acts are value binder in a similar way as in a lambda term. The agent \(A\) will behave as \(A[v/x]\) after the input \(v\) is received through port \text{in}.

- Names in an output action have an applied nature, and do not introduce scope regions. They are just identifiers in a value expression, conforming to a call by value semantics.

The initial value of the value part of the memory cell cannot be specified within the agent itself. It is set through the behaviour expression that introduces the agent, which is syntactically a form of the *AgentService*\(^{PS}\) class. It resembles the function call in the value part of *MOSCA*. A agent service construction has two parts, the agent name it invokes, and an actual value for the value part, if present of course. The expression

\[
\text{MCell}\ (0)
\]

is an agent service expression that

- sets the value part of the agent *MCell* to 0, and

- behaves just as the behaviour expression in the agent definition for *MCell* that covers the actual value 0.

**Example 3.3** The next specification uses sequence patterns and gives a model for a simple unbounded buffer that buffers natural numbers.
3.2. Agent definition

\[
\begin{align*}
\text{Buffer } (N^*) & \\
\text{ports in in : N} & \\
\text{out out : N} & \\
\text{Buffer } ([]) & \triangleq \text{in(z) \circ Buffer } ([x]) & \\
\text{Buffer } ([x] \triangleright s) & \triangleq \text{inout(x) \circ Buffer } (s) \oplus \text{in(y) \circ Buffer } ([x] \triangleright s \triangleright y)
\end{align*}
\]

The agent Buffer has a single value part that can hold sequences of natural numbers, \(N^*\). It offers two ports, one input port in that accepts a number, and an output port out that delivers a value from the buffer. The sequence patterns ‘[]’, and ‘[x] \triangleright s’ are used to mark the two cases of behaviour for the buffer. An empty buffer can only accept values at the input port, whereas a non-empty buffer may accept a value for buffering and may deliver a value from the buffer to the surrounding system. The two sets of values matching these two patterns are clearly disjoint assuring deterministic behaviour.

An alternative way for specifying the unbounded buffer from example 3.3 is given in spec. 3.6. Here only one behaviour definition is used. It applies a

\[
\begin{align*}
\text{Buffer } (N^*) & \\
\text{ports in in : N} & \\
\text{out out : N} & \\
\text{Buffer } (s) & \triangleq \text{if } s = [] \text{ then in(z) \circ Buffer } ([x]) \text{ else inout(hd } s) \circ Buffer \ (\text{tl } s) \oplus \text{in(x) \circ Buffer } (s \triangleright [x])
\end{align*}
\]

sequence pattern that matches all possible values of the value part. The case analysis applied in example 3.3 by using multiple patterns is here replaced by the usage of the AgentIf\(P^S\) expression, in combination with the explicit usage of the sequence operations hd and tl.

The type of the value part can be any mosca value type. In the MCell example I used a basic type, \(Z\). In the agent Buffer I used a sequence type. In the next example I will apply a product type. In specification 3.7 the Buffer agent is given a counting device that holds the maximum number of elements ever in the buffer. The agent CBUFFER offers three ports: one input and output port similar to the ports of the agent Buffer, and one
Chapter 3. Notation Overview

output port that produces the number that states the maximum length of the buffer during its lifetime so far. The type of the value part is now a product type, \( \langle \mathbb{N}^* \times \mathbb{N} \rangle \). The first part states the type of the value holding the buffered elements, the second part gives the type of the counting device. The agent behaviour constructions for \emph{CBuffer} use product patterns. The

\[
\begin{align*}
\text{CBuffer} (\langle \mathbb{N}^* \times \mathbb{N} \rangle) & \triangleq \\
\text{ports in } \text{in} : \mathbb{N} \\
\quad \text{out } \text{out} : \mathbb{N} \\
\quad \text{out } \text{cnt} : \mathbb{N} \\
\text{CBuffer} (\langle [], \text{max} \rangle) & \triangleq \\
\quad \text{in}(x) \odot \text{CBuffer} (\langle [x], \text{if max} > 0 \text{ then max else 1} \rangle) \\
\quad \oplus \text{cnt}(\text{max}) \odot \text{CBuffer} (\langle [], \text{max} \rangle) \\
\text{CBuffer} (\langle [x] \leadsto s, \text{max} \rangle) & \triangleq \\
\quad \text{out}(x) \odot \text{CBuffer} (\langle s, \text{max} \rangle) \\
\quad \oplus \text{in}(y) \odot \text{let newbuf} = [x] \leadsto s \leadsto [y] \text{ in} \\
\quad \quad \text{if len newbuf} > \text{max} \\
\quad \quad \text{then CBuffer} (\langle \text{newbuf}, \text{len newbuf} \rangle) \\
\quad \quad \text{else CBuffer} (\langle \text{newbuf}, \text{max} \rangle) \\
\quad \oplus \text{cnt}(\text{max}) \odot \text{CBuffer} (\langle [x] \leadsto s, \text{max} \rangle)
\end{align*}
\]

first behaviour construction shows that an empty \emph{CBuffer} may accept an element for buffering,

\[
\text{in}(x) \odot \text{CBuffer} (\langle [x], \text{if max} > 0 \text{ then max else 1} \rangle)
\]

and may offer the length so far through

\[
\text{cnt}(\text{max}) \odot \text{CBuffer} (\langle [], \text{max} \rangle)
\]

The two alternatives are separated by the choice operator \( \oplus \). The construction applied to compute the counter after the input action is an ordinary value let construction. In the second behaviour expression the value of the counter is set using an agent if construction.

In the body of the second agent behaviour a \emph{AgentLet}^P^S construction is used that introduces the name ‘newbuf’ as a shorthand for the buffered result of the prefix ‘\emph{in}(y)’. An agent if expression is used to compute the correct buffer length, which is either the length of the previous buffer \emph{max}, or the length of the new buffer contents newbuf.
3.2. Agent definition

In MOSCA, like in VDM, functions can be specified explicitly and implicitly. In specification 3.8 we present a combination of an agent with an implicitly specified function.

Example 3.4 Let’s give an additional requirement for the buffer specification given in example 3.3. The values in the buffer are now being kept in a sorted order. Each new value is added in such way that the ordering in the buffer is preserved. The largest value is being offered as output. The agent

\[
SBuffer (N^*)
\]

ports in \(in\) : \(N\)

out \(out\) : \(N\)

\[SBuffer \ (s) \triangleq \begin{cases} \text{if } s = [] & \text{then } in(x) \odot SBuffer \ ([x]) \\ \text{else } out(hd \ x) \odot SBuffer \ (tl \ x) & \odot in(x) \odot SBuffer \ (Sort(x, s)) \end{cases}\]

where

\[Sort \ (x:N_s, s:N^*) \ r:N^*\]

post let \(sz = s \prec [x]\) in

\[\text{is\_ordered}(r) \land \text{is\_permutation}(r, sz)\]

\(SBuffer\) contains a local function definition, following the keyword ‘where’. The function \(Sort\) employs two predicates, \(\text{is\_ordered}\) and \(\text{is\_permutation}\), which are assumed present. The predicate \(\text{is\_ordered}\) guarantees that the elements in the sequence are sorted in decreasing order. The predicate \(\text{is\_permutation}\) guarantees that its two arguments hold the same elements.

\[\square\]

3.2.3 Type II Agents

The main difference between the Type I and Type II agents is the way local values are handled. Type I agents are equipped with value parts that can be handled in a functional fashion. The ports specification can be followed by a full VDM-SL state specification constructed as a state type definition, a state invariant and a state initial function. After the behaviour specifications a series of local definitions may be given, including operations that may act upon the agent state. These operations can only be applied within the behaviour specifications of the agent.

State initialization is now achieved through the state initial function which states the condition for the state value that holds on the activation
of the agent. Through a combination of state and value parts state-values can be passed to other agents. Specification 3.9 presents a typical MOSCA Type II agent. The state values are re-initialized on each Type II agent-

\[
\begin{align*}
\text{Agent.name} \triangleq & \quad \text{invariant.function.definition} \\
\text{ports in importname} & : \text{Type} \\
\text{out outpostname} & : \text{Type} \\
\text{state var1} & : \text{Type1} \\
\text{var2} & : \text{Type2} \\
\ldots & \\
\text{varn} & : \text{Typen} \\
\text{inv-Agent.name(state_pattern)} & \triangleq \text{invariant.function.definition} \\
\text{init-Agent.name(state_pattern)} & \triangleq \text{invariant.function.definition} \\
\text{Agent.name} & \triangleq \text{Agent.Behaviour.Expression} \\
\text{where} & \\
\text{Operation (arg1: type1, \ldots, argn: typen) res: typeres} & \\
\text{ext rd var1} & : \text{Type1} \\
\text{wr varn} & : \text{Typen} \\
\text{pre \ldots} & \\
\text{post \ldots} & 
\end{align*}
\]

service application that requires an agent scope switch. The state values are retained when the agent denoted in the agent-service application is the same as the agent in which the agent-service application occurs, like in cyclic agent behaviour definitions.

State manipulation is only possible through operation calls or statements within the context of a state manipulation, which is a special form of a prefix construction. The notation

\[
\sigma(\text{OPCALL(arg1, \ldots, argn)}) \odot P
\]

is an example of a state manipulation. The action is a special action, with only internal effects, that is not visible outside of the agent. Its effect is the application of the specified operation, thus possibly resulting in a changed state-value. So all state-changing computations are performed
3.2. Agent definition

within actions of prefix constructions. State manipulation can never be interrupted, and run always to completion before any other action may be performed. State values are never shared between agents. An agent encapsulates its private state.

Example 3.5 (Sorted buffer with local state) In specification 3.8 the contents of the buffer was modelled through a value part. Here I have modelled it as a state component. The VDM-SL operations ADD, DELHD

\[
SBuffer
\]

\[
\begin{align*}
\text{ports in } & \text{in : } N \\
\text{out } & \text{out : } N \\
\text{state } & \text{contents : } N^* \\
SBuffer & \triangleq \\
\text{if contents } = [] & \text{then } \text{in}(x) \circ \sigma(ADD(x)) \circ SBuffer \\
& \text{else } \text{out}(\text{hd } x) \circ \sigma(DELHD()) \circ SBuffer \\
& \quad \odot \text{in}(x) \circ \sigma(ADD(x); \text{SORT}()) \circ SBuffer
\end{align*}
\]

and \textit{SORT} are assumed to be specified in the context of the agent. \hfill \square

Agents can be visualised through dedicated pictorials, for each style of definition. Figure 3.2 displays some examples. These pictorials have similar forms as the ones used by MILNER in [157]. Ovals mark the scope regions

![Pictorial Examples]

Type 0 agent

Type 1 agent

Type II agent

Figure 3.2: Typical agent pictorials

created by agent definition, small black blobs mark the ports, circles within the agents mark the value parts and rectangles mark the local state. These pictorials come to serve nicely when dealing with complex expressions containing compositions.
3.2.4 Mixed Agents

Agents may apply both value encapsulation techniques at the same time. These agents are called mixed agents. The value part may serve as interface to the environment as alternative to value passing through synchronization between two agents in input actions. The state part may be used as internal data container serviced by state manipulating operations. The value part can also be used to pass state values from one agent to another.

Example 3.6 Typical data moves are depicted in specification 3.11. The

| \[ MAGI \quad \text{state } data : N^* \] |
| \[ MAGI \triangleq \ldots \odot \sigma(\text{COMPUTE}(\ldots)) \odot MAGO (data) \] |
| \[ MAGO (N^*) \quad \text{state } data : N^* \] |
| \[ MAGO (v) \triangleq \ldots \] |

first agent MAGI computes a sequence of naturals and passes this value to a successor agent MAGO through its value part.

This technique applies e.g. when values must be passed to agents that continue the computation. If the sending agent remains active then value passing communication can be applied.

3.3 Agent Behaviour

In this section I will enumerate the constituents of behaviour expressions of MOSCA. Figure 3.3 presents a summery of the behaviour expressions in MOSCA. It gives for each presented syntactical entity a typical example. The first block is dedicated to agent instantiation. The second block contains various forms of if and let expressions.

- The AgentIfPS expression enables conditional behaviour specification. The AgentValueLetPS and AgentValueLetBePS are very powerful constructions.

- Through recursive let constructions very complex agents can be defined in an elegant way.

- The AgentAgentLetPS construction enables local agent definitions.
3.3. Agent Behaviour

<table>
<thead>
<tr>
<th>Construction</th>
<th>Typical Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 0 AgentService</td>
<td><em>Agent</em></td>
</tr>
<tr>
<td>Type 1,2 AgentService</td>
<td><em>Agent (v)</em></td>
</tr>
<tr>
<td>AgentIf</td>
<td>if Condition then Agent_1 else Agent_2</td>
</tr>
<tr>
<td>AgentValueLet</td>
<td>let pattern = expr in Agent</td>
</tr>
<tr>
<td>AgentValueLetBe</td>
<td>let pattern ∈ Set in Agent</td>
</tr>
<tr>
<td>AgentAgentLet</td>
<td>let identifier = Bexpr in Agent</td>
</tr>
<tr>
<td>Timed syn Prefix</td>
<td>syn_act, *t ⊗ Agent</td>
</tr>
<tr>
<td>Timed in Prefix</td>
<td>in_act(pattern), *t ⊗ Agent</td>
</tr>
<tr>
<td>Timed out Prefix</td>
<td>out_act(value), *t ⊗ Agent</td>
</tr>
<tr>
<td>StateManipulation</td>
<td>σ(statement) ⊗ Agent</td>
</tr>
<tr>
<td>Choice</td>
<td>Agent_1 ⊕ Agent_2</td>
</tr>
<tr>
<td>Composition</td>
<td>Agent_1</td>
</tr>
<tr>
<td>Restriction</td>
<td>Agent \ {label1, ..., labeln}</td>
</tr>
<tr>
<td>Relabelling</td>
<td>Agent[I1 → I'1, ..., ln → l'n]</td>
</tr>
<tr>
<td>Null Agent</td>
<td>null</td>
</tr>
</tbody>
</table>

Figure 3.3: mosca Behaviour expressions

The next block is dedicated to the various forms of \( \text{Prefix}^{PS} \) expressions. The first three are regular CCS forms extended with an explicit synchronization prefix. The fourth item handles the infusion of value state manipulation. It specifies a VDM-SL statement that may manipulate the state of the agent in which the construction appears.

The fourth block contains standard CCS operators.

* The \( \text{Choice}^{PS} \) construction enables nondeterministic selection of specific behaviour.

* \( \text{Composition}^{PS} \), and \( \text{Restriction}^{PS} \) involve agent communication.

* Through \( \text{Relabelling}^{PS} \) port labels can be renamed.

They originate in CCS and have a meaning equivalent to their CCS counterparts. The last block provides a means to specify a totally inert agent, not capable of any observable action. It is denoted by the reserved word null.

3.3.1 Instantiation and Termination

The basic form of agent instantiation is the \( \text{AgentService}^{PS} \) construction. This construction is equivalent with the agent constant expression of CCS.
The behaviour of an agent service is equal to the behaviour expression associated with the agent name in the agent definition. For agents defined by multiple behaviour expressions, the actual value of the agent value part selects the associated variant. Given e.g. the specification in example 3.3, the agent service expression \emph{Buffer ([[]])} would select the first behaviour expression, and \emph{Buffer ([[1, 2, 3]])} would select the second behaviour expression.

Until so far we have only seen agent definitions with infinite behaviour. This infiniteness has been a direct result from cyclic behaviour expressions, like

\[ MCell \langle val \rangle \triangleq \text{set}(x) \circ MCell \langle x \rangle \]

After finishing the action \text{set} the behaviour of the agent \emph{MCell} is fixed by the agent service expression \emph{MCell} \langle x \rangle, which results in starting all over again, etc. If we want to model agents with a finite behaviour, other description techniques than cyclic descriptions are needed. Here the usage of the \emph{NullAgentPS} expression applies. Its syntactical representation is given by the reserved word \emph{null}. Its behaviour is void, i.e. the agent is completely inert. A null agent cannot be engaged in any simultaneous actions, has no state nor any value parts. From an external point of view, we say that a transition that transforms an agent behaviour expression into a null agent \emph{terminates} the agent. Consider e.g. agent definition 3.12, that specifies a variant of the \emph{MCell} agent, that is terminatable. The capabilities

| \multicolumn{2}{c}{3.12 Terminatable agent} |
|-----------------------------|-----------------------------|
| \text{ports in set} \colon \text{Z} | \text{out \text{get}} \colon \text{Z} |
| \text{syn \text{stop}} | |

\[ TMCell \langle \text{val} \rangle \triangleq \text{set}(x) \circ TMCell \langle x \rangle \oplus \text{get}(\text{val}) \circ TMCell \langle \text{val} \rangle \oplus \text{stop} \oplus \text{null} \]

of the agent \emph{TMCell} are equivalent to the agent \emph{MCell}, with the difference that \emph{TMCell} is terminatable through the port labelled \text{stop}.

### 3.3.2 Composition, Relabelling and Restriction

Composition, relabelling and restriction are three CCS operations that act together in the description of communicating agents. In Milner's recent book on CCS [157] these three operators are called \emph{static} with respect to
their effect within behaviour expressions. Under control of the operational semantics they never disappear once present in a behaviour expression. MILNER uses his equational characterization to eliminate useless constructions.

Communication between agents is enabled by linking agents together through the \emph{Composition} construction. Given two agent behaviour expressions \( E_1 \) and \( E_2 \), then their composition is given by the behaviour expression \( E_1 \mid E_2 \). The composition combinator \( \mid \) links the two agent expressions through ports that have the same label, where,

- input ports are linked with output ports, and
- synchronisation ports are linked with synchronisation ports.

In the first case, the connection enables a communication through which values are passed from output ports to input ports. The second form of linkage gives a communication consisting of pure synchronisation between the involved agents. Communication results from prefix expressions.

Let's consider again the memory cell given in specification 3.4. This agent could be part of a system in which several other agents would use the memory cell for reading and writing. We could specify such an user as in spec. 3.13. The \emph{User} agent somewhere in its behaviour wants to read the

\[
\text{User (}Z \rightarrow Z\text{)}
\]

<table>
<thead>
<tr>
<th>ports in \text{read} : ( Z )</th>
<th>3.13 User agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>out \text{write} : ( Z )</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{User (action)} \triangleq \text{read}(x) \odot \text{write(action(x)))} \odot \ldots
\]

value of the memory cell, computes a replacement value through a generic function, supplied through the agent service construction, and writes the latter back into the memory cell. In the sequel some of these functions, \( f, g \) are assumed. To do so it needs a connection through its \text{read} port to the \text{get} port of the memory device, and a connection through its \text{write} port with the \text{set} port of the memory cell. To set up a system, composed by the \emph{MCell} agent and the \emph{User} agent, the composition

\[
\text{MCell} \mid \text{User (}f\text{)}
\]

alone is not sufficient. As stated above, the composition operator \( \mid \) links up ports with complementary names. But there are no complementary names in the \emph{MCell} \( \mid \) \emph{User} expression. So the \( \mid \) operator will not introduce any
connections between the two agents. They act merely in parallel without any means to communicate. To create a connection in this case we need the Relabeling PS construction. A relabeling is a mapping from port labels to port labels, that renames specific port labels from a given agent. This relabeling influences only the port labels in the scope space external to the agent. Within the agent itself, the port names are constant.

The relabeling of the User agent has the following syntactical representation:

\[ \text{User } (f) \ [\text{read } \mapsto \text{get}, \text{write } \mapsto \text{set}] \]

The general form consists of a behaviour expression, followed by a list of relabelling maplets.

The connection between the two agents does not prohibit other agents to engage in the same communication. Consider another User agent, equal to the first User, that is also linked to the MCell agent through composition, e.g. in:

\[
\text{let } \text{RLUser1 } = \text{User } (f) \ [\text{read } \mapsto \text{get}, \text{write } \mapsto \text{set}] \\
\text{RLUser2 } = \text{User } (g) \ [\text{read } \mapsto \text{get}, \text{write } \mapsto \text{set}] \text{ in} \\
(\text{RLUser1 } | \text{MCell}) \ | \text{RLUser2}
\]

The | operator is associative, that is, \( A \ | \ (B \ | \ C) \) has the same meaning as \( (A \ | \ B) \ | \ C \), as well as commutative, that is, \( A \ | \ B \) behaves equally as \( B \ | \ A \). Both RLUser agents will compete for the usage of the MCell agent. Each time the MCell agent is ready for a read action, only one of the two RLUser agents can be involved. But after a read action, the MCell can either accept another read, or perform a write action. So it can happen that a simultaneous read action between MCell and the left RLUser is followed by a simultaneous write action between MCell and the right RLUser agent.

From the user’s point of view, this is clearly an unwanted situation. It can be solved in different ways. As a first solution, we could adapt the behaviour of the MCell agent to a form in which read and write actions are performed in a cyclic way, each read action followed by a write action. This may be acceptable for the user, but totally unacceptable as behaviour of the memory device itself. It would be a severe constraint to demand cyclic read-write actions.

A second solution introduces an arbitrating device, that regulates the entrance to the memory device. Such a device is modelled easily as a semaphore agent, such as the agent given in specification 3.2. We only need to change the behaviour of the RLUser agents to incorporate requests to the semaphore agent, followed by accessing the memory device. This adaptation is done in spec. 3.14
3. Agent Behaviour

\[ PVUser \ (Z \rightarrow Z) \]
\[
\text{ports in } \underline{\text{read}} \ : \ Z \\
\underline{\text{out write}} \ : \ Z \\
\text{syn } \ p \\
\text{syn } \ v \\

PVUser \ (\text{action}) \ \triangleq \\
p \odot \underline{\text{read}}(z) \\
\odot \underline{\text{write}}(\underline{\text{action}}(val)) \odot \ v \\
\odot \ldots
\]

3.14 Adapted User Agent

The agent behaviour specification for the whole system is presented in spec. 3.15. \(^1\) The interface of the agent is fully determined by the interfaces of the agents in the behaviour expression \(User1 \mid MCell \mid User2 \mid Semaphore\) and is subsequently omitted.

\[ System \ \triangleq \]
\[
\begin{align*}
\text{let } User1 &= PVUser \ (f) \ [\text{read } \leftrightarrow \text{ get, write } \leftrightarrow \text{ set}] \text{ in} \\
\text{let } User2 &= PVUser \ (g) \ [\text{read } \leftrightarrow \text{ get, write } \leftrightarrow \text{ set}] \text{ in} \\
User1 \mid MCell \mid User2 \mid Semaphore
\end{align*}
\]

3.15 System

The last operation involved with communication is the \(Restription^{PS}\) operator ‘\(\backslash\)’. Its effect is to hide ports in the interface of agents. These ports become inhibited from partaking in any communication. E.g. given the original \(User\) definition, the expression

\[ User \ (f) \ \backslash \ \{\text{read}\} \]

disables the read port in the \(User\) agent. Internally the read port still exists, the ‘\(\backslash\)’ operation does not delete ports, it only restricts the port from communication.

A memory cell dedicated solely to the service of a particular user could be specified using the restriction operator:

\[(MCell \mid User \ (f)) \ \backslash \ \{\text{set, get}\}\]

that is, the system \(MCell \mid User \ (f)\) is further restricted from any usage of the ports that are linked up. The restriction does not effect linking that

\(^1\)The parenthesis may be omitted in the behaviour expression \(User1 \mid MCell \mid User2 \mid Semaphore\), resulting from the associative behaviour of the \(|\) operator.
results from the $MCell \mid User$ expression. The agents $MCell$ and $User$ are not influenced by the restriction, only the combination is restricted. That is, the restriction does not distribute over composition. So, the following inequality will hold:

$$(A \mid B) \setminus \{x\} \neq A \setminus \{x\} \mid B \setminus \{x\}$$

where $A$ and $B$ are arbitrary agents, and $x$ is a common port to both $A$ and $B$.

### 3.4 Units, Collections, Scopes and Visibility

Units are the only construction in Mosca capable of modelling shared memory devices. Collections offer a primitive structuring mechanism for Mosca specifications. The section is closed with a review of the scope rules in Mosca.

#### 3.4.1 Units

A unit definition defines one agent for which the heading and port specifications may vary during its lifetime. Consider the agent $AgentA$ in spec. 3.16. Its interface consists of two ports input and output. It has no value part.

<table>
<thead>
<tr>
<th>unit $AgentA$</th>
<th>3.16 Agent with variable interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>ports in input : $N$</td>
<td></td>
</tr>
<tr>
<td>out output : $N$</td>
<td></td>
</tr>
<tr>
<td>shares $AgentB \ (N)$</td>
<td></td>
</tr>
</tbody>
</table>


$$AgentA \triangleq \text{input}(v) \circ AgentB \ (v)$$

$$AgentB \ (value) \triangleq \text{compute}(value) \circ AgentA$$

It changes after acceptance of an integer value to an execution state $AgentB$ which applies a value part. The value part of $AgentB$ is not available to the environment. The unit $AgentA$ is activated by the normal agent-service construction. $AgentB$ is not activatable from the environment, only from within the agent $AgentA$ itself.

The interface of the constituents of the unit inherits all elements of the initial execution state, the name barer of the unit. Each execution state may extend the inherited interface. The state of the unit is shared among all execution states. It is not extendable. On activation of a successor
3.4. Units, Collections, Scopes and Visibility

execution state the state values are retained. Through the composition of successor execution states it is possible to model shared memory.

Example 3.7 (Shared Buffer) Specification 3.17 models a message buffer capable of holding single messages up to 1KB, able to service two reading and writing clients in parallel. The buffer models a shared mailbox of capacity 1. The unit $SHBuffer$ does not define a shared interface. Each execution state has its private interface defined within the share list. The unit does define a shared value state $message$, modelled as a sequence of characters. The share list defines next to the initial execution state of the $SHBuffer$ two additional execution states, Reader and Writer, each with a private interface. The behaviour expression for the initial execution state

$$Reader | Reader | Writer | Writer$$

offers the merged interface of both Reader and Writer, execution states to the environment, and defines an agent capable of servicing two reading or writing requests in parallel. Suppose two clients are communicating by message passing and apply a shared message buffer as communication medium. Their composition would become

$$Client1 | SHBuffer | Client2$$

and their pictorial representation is sketched in figure 3.4. The internal

Figure 3.4: Shared buffer with two clients

structure of the two clients should offer a communication protocol over the usage of the buffer such that a meaningful application is possible. Without
such protocol messages can be lost. Messages can clearly be overwritten by new messages without ever being read. These new messages may originate from both clients. To avoid complicated protocols the buffer can be realized with two memory devices, one for incoming messages for client1 and another one for client2. Further, the capacity of the mailbox can be enlarged to hold a number of messages instead of one.

3.4.2 Collections

There is no strong structuring mechanism in MOSCA. A module structure is not present. The reason for this omission is simply the problem of giving a strong module mechanism a meaningful semantics. BEAR proposes in [24] a module structure with parameterized modules, polymorphic arguments, import and export of types, functions, operations and state. After much discussion within the BSI/VDM panel the structuring is omitted from the VDM standard in development, just for semantical reasons. It has not been possible to construct a meaningful semantics for modules within the framework of the overall semantic definition of VDM-SL, which is explained in the next chapter. MIDDELBURG has provided in [152] a structuring mechanism for VVSL, another extended form of VDM-SL. He has defined a complete different semantics for the whole language.

The only structuring notion is the encapsulation of a set of definitions through a collection. A collection consists of an static set of items. Once defined, they cannot be changed.

Items have unique names. An item may consist of the definition of a type, value, function, agent or unit. As such, collections can be completely (i) value-oriented, e.g. ADT's, (ii) agent-oriented, e.g. built from server agent definitions or (iii) mixed. Notice that collections cannot define state. Collections are flat, i.e. they cannot import anything.

The outermost scope construction in MOSCA is a specification. A specification is built from (i) a series of unit-import clauses, (ii) a list of local definitions and (iii) a specification-expression, i.e. an agent, in which all agent and unit definitions defined in the specification can be composed. Items can be imported into specifications by either explicit item import constructions, or collection import resulting in all items imported together.

Collections form a pure syntactical notion. In the core abstract syntax of MOSCA, given in appendix B, the top level node is Spec, and all defined entities resulting from import and local definitions are merged into one set of definitions.
3.4. Units, Collections, Scopes and Visibility

```plaintext
unit SHBuffer
state message : Char*

shares Reader
    ports out get_message : Char*
    Writer
    ports in set_message : Char*

inv-message(s) ⊢ len s = MAX_MEMORY
init-message(s) ⊢ ∀i ∈ inds s · s(i) = 0

SHBuffer ⊢ Reader | Reader | Writer | Writer

Reader ⊢ get_message(v) ◦ Reader
Writer ⊢ set_message(v) ◦ σ(UPDATE(v)) ◦ Writer

where
values
MAX_MEMORY : N = 1024
end

UPDATE (v : Char*)
ext wr message : Char*
post if len v ≤ len message
    then ∀j ∈ len v · message(j) = v(j) ∧
        ∀i ∈ (inds message \ inds v) · message(i) = message(i)
    else message = message
```

3.17 Shared Buffer
3.4.3 Scope and Visibility

Each scope region marks the borders of the effect of a definition. The scope of a defined entity is the range of the MOSCA constructions over which the entity is known and manageable.

MOSCA is a flat language. The scope introducing entities are specifications, agent or units, agent behaviour definitions and behaviour expressions like prefix, value-let and agent-let expressions.

- The outermost scope level is effectuated by the specification structure, that encapsulates value defining constructions like types, constants and functions, and may contain agent and unit definitions.

- The agent (or unit) constructs the succeeding scope level. Agents encapsulate types, constants, functions, operations and a state definition. Further they name ports.

- Agent behaviour expressions and unit execution states introduce the next scope level. Here we may find additions like (i) names bounded to values through value-part pattern matching in both structures and (ii) additions of port names due to execution states only.

- Behaviour expressions, like prefix- and let-expressions define the innermost scope level of MOSCA.

The nested scopes are visualised in figure 3.5.

Example 3.8 Let us look at the scopes of the different names in spec. 3.5. The variable names in an input action value pattern have a restrictive scope, namely the prefix expression in which they occur. As such in \( \text{in}(x) \circ \text{Buffer } ([z]) \), the 'x' is bound by the prefix \( \text{in}(x) \).

When we mark the different scopes in the second agent behaviour construction of specification 3.5 we get the following.

\[
\begin{align*}
\text{Buffer } ([x] \xrightarrow{\circ} s) & \triangleq \underbrace{\text{out}(x) \circ \text{Buffer } (s) \oplus \text{in}(y) \circ \text{Buffer } ([x] \xrightarrow{\circ} s \xrightarrow{\circ} [y])}_{\text{scope } x,s}
\end{align*}
\]

The visibility of an entity is basically depending of the scope of the entity. Inside its scope the entity is visible, outside its scope it is not visible. The visibility of constructions within specifications is in general an important aspect of the structure of the total name-space. The visibility of
entities is influenced by both static and dynamic influences. Static influences are e.g. the scope-region changes, dynamic influences are e.g. port renaming through renaming expressions. MOSCA has no strong visibility rules like e.g. the programming language Ada ([217]). Van Katwijk has collected within his thesis [122] an overview of scope and visibility aspects of programming languages, in particular Ada, and provides implementation schemes. In MOSCA the visibility of entities in encapsulating scope regions is influenced by entities in the encapsulated scope regions. Equal names may be used for different notions (overloading) like in the Ada languages and for equal notions in different scopes (hiding). The VDM-SL part is governed by the static semantic rules of VDM-SL, recorded in the ISO standard to be. An early presentation of these static semantic rules can be found in [46] and [60].

The MOSCA constructions dedicated to describe the concurrency and time-dependent behaviour behave in accordance with the semantic rules given in appendix C. As is depicted graphically in figure 3.5 the different
scope regions appear nested within each other. Names defined in inner
scope regions hide notions carrying equal names and equal types in encapsu-
sulating scope regions.

Example 3.9 (Hiding) The next behaviour specification presents some
examples of overloading and hiding.

\[
\text{Buffer } \langle [\text{Buffer}] \sim s \rangle \triangleq \\
\text{out(}\text{Buffer}) \circ \text{Buffer } (s) \oplus \text{in}(s) \circ \text{Buffer } \langle [\text{Buffer}] \sim s \rangle
\]

The name Buffer is used both as agent name and as name for a data element
in the buffer, which is admissible (but certainly not a good presentation
style) in MOSCA. As such the name is overloaded and refers to different
entities in different contexts. It is not possible to overload a name with
either two VDM-SL notions or two notions from the process defining part of
the language, even in different contexts. Using the same name for equivalent
notions in different scope regions results in hiding. The name s used as
input value pattern name hides the binding introduced by the usage of s in
the value part expression of the agent behaviour definition in a permanent
way.

The only entity that has a variable visibility is the port name. By renam-
ing a port name may have different names in different scopes. Application
of the restriction operator inhibits the visibility of ports outside of the scope
of the expression in which the operator is applied.

This section has presented some (and certainly not all) properties of
the static aspects of the name-space of MOSCA. The effect of a defining
occurrence of a name is the realization of a binding of the name with the
semantic notion it represents. The dynamic properties of the name-space
are related to the extend of the bindings. In

\[
\text{Buffer } \langle [z] \sim s \rangle \triangleq \ldots
\]

The name z denotes a sequence value. The name is bound to a value as
a result of an agent service of the Buffer agent in a behaviour expression.
The extend of the binding (also denoted as its lifetime) is dependent of
the nature of its context. In the next chapter the dynamic properties of
bindings are treated separately.

3.5 Time specification

This section contains a presentation of the constructions in MOSCA that
handle time. Section 3.5.1 presents a summary of all constructions. The
3.5. Time specification

next three subsections handle the specific constructions in more detail. The last subsection in this section finalises the series of buffer specifications by presenting two versions of timed buffers.

3.5.1 Overview

The model of time is centered around the following issues.

- There is no central clock, or any other ticking device that registers the current time.

- Passing of time is measured related to actions: from the moment an action can be taken, i.e. offered to the environment, to the moment the action is actually performed in a communication. The passed time is recorded in time variables associated with the involved action.

- Synchronization actions are timeless. The actual taking of input, output or state manipulating actions can either be timeless or may consume a specific amount of time.

- Time progression results from taking time consuming actions or by idle actions.

The model approach to time is strongly influenced by Wang C.S., Chen C.S and Moller C.S.. Their model fits in most closely with the functional approach of CCS itself. Time identifiers of timed prefix constructions may occur as free variable within the behaviour expression following the $\odot$ operator. The effect of the prefix construction substitutes the actual value of time variable $t$ in each free occurrence of the identifier $t$ within the behaviour expression in $P$. This approach is similar to $\beta$ substitution within the various forms of the $\lambda$-calculi, where a $\lambda$-term $\lambda z.f$ computes a value by substituting an actual value $v$ in each free occurrence of $z$ in $f$. Thus in

$$\text{act, } \star t \odot Bexpr$$

each free occurrence of $t$ in $Bexpr$ is bound by $\star t$. Upon taking the action $\text{act}$ the actual value of the time variable $t$ is substituted in all free occurrences of $t$ in $Bexpr$.

Time progression is due to the taking of idle actions. E.g.

$$\text{idle}(5.2) \odot P$$

specifies a behaviour expression, that causes the time to progress with the value 5.2. The time progression is strongly connected to the semantics of
the prefix construction, choice and composition operator. Its semantics is treated in detail in section 4.5.

Synchronization actions take no time. The taking of input and output actions may take time to proceed. To model the time consumption due to value computation and passing, the input and output actions have extended forms that hold besides the involved data a time value.

\[ \text{input}(\text{data}, \ast e), \ast t \odot P \{ t \} \]

models an agent waiting for \( t \) time units to take the input action. The actual computation and passing of the value \( \text{data} \) takes \( e \) time units. Here \( e \) is a time expression, delivering a fixed time value. Dually, output actions may take an equal amount of time. In

\[
\ldots \text{input}(\text{data}, \ast e_1), \ast t_1 \odot \ldots \mid \\
(\ldots \text{output}(\text{data}, \ast e_2), \ast t_2 \odot \ldots ) \backslash \text{[output} \leftrightarrow \text{input}] 
\]

\( t_1 \) and \( t_2 \) will deliver independent values, set by the evolution of the system, but \( e_1 \) and \( e_2 \) must be set by specification. State manipulations are equally specifiable through an extended form, and then considered to occur by consuming time. E.g. to model the time consumption of 10 ms by a state manipulation through operation \( UPDATE \) after a timeless acceptance of a computation data \( d \) can be modelled as (assuming an one second time scale)

\[ \text{input}(d) \odot \sigma(UPDATE(d), \ast 0.01) \odot \ldots \tag{3.1} \]

Many of the approaches described in the literature refrain from the possibility of time consumption by ordinary action. They start from the idea that to model this kind of time consumption idle actions should be used. This does not reflect reality as it forces the specifier to introduce specific time-consuming actions (i.e. idle actions) to model the time consumption of computation steps, without retaining a semantic connection between the two constructions. Consider again the former state manipulation. Following the latter approach we get e.g.

\[ \text{input}(d) \odot \sigma(UPDATE(d)) \odot idle(0.01) \odot \ldots \tag{3.2} \]

in which no semantic connection between the state manipulation and the idle action is maintained. So the appropriate intuition for expression 3.2 would be that the agent takes the input action, performs the \( UPDATE \) operation and then idles 0.01 seconds, which clearly deviates from the intuition for expression 3.1 which holds that after taking the input action the agent performs the \( UPDATE \) operation by consuming 0.01 seconds.
3.5. Time specification

The domain \( T \) for the values of the idle actions is chosen to be \( \mathbb{R}^+_0 \),
the set of real values greater or equal to zero. Default the following type
definition applies:

\[
T = \mathbb{R} \\
inv(t) \triangleq t \geq 0
\]

The set of reals creates a model that fits more closely to reality than e.g. \( \mathbb{N} \),
the natural numbers. A time action is not the same as the normal actions
(which are fully controllable) since the latter ones may be prevented from
happening by the environment and time progression can never be blocked.
The parallel composition is synchronous with respect to time events. E.g. in

\[
idle(6) \odot P \mid idle(8) \odot Q
\]
time may progress e.g. with 2 time units, resulting in

\[
idle(4) \odot P \mid idle(6) \odot Q.
\]

However, time progression is asynchronous for the normal actions except
communications between its different components. In

\[
P \mid Q
\]

\( P \) and \( Q \) can perform actions asynchronously until a synchronization be-
tween the two subcomponents is due. For a fully synchronous system one
may take a discrete time domain (like with SCCS in [156]), since all sub-
agents refer to the same global intuition of time (e.g. a global clock), and
events happen at certain moments in time. In the asynchronous case, any
two agents may perform actions at times that are not equal, but arbitrary
close to each other. Hence a dense time domain is needed.

In MOSCA the time model may be used defined to cater for maximum
flexibility. Hence, the type definition

\[
T = \mathbb{N}
\]

redefines the time domain to encompass the natural numbers only.

3.5.2 Timed Prefix

Time enters the specification through the prefix construction. All in-, out-, and
synchronization prefixes are extended to timed prefixes. The timed pre-
fix consists of an action and a behaviour expression, specifying replacement
behaviour. The registration of time progression is achieved by extension
of the standard prefix construction to a *timed* prefix construction: that contains a time variable, e.g. \( *t \). In

\[
a, *t \odot P
\]

\( t \) is a time variable bound to the action \( a \). It will register the progression of time from the moment action \( a \) is offered to the environment to be taken, to the moment the action is accepted by the environment. E.g. when the environment accepts action \( a \) after 4.2 time units, the value of time variable \( t \) will be 4.2. This value is substituted for each free occurrence of \( t \) in \( P \).

### 3.5.3 Idle Action

There is a special action which specifies a certain amount of time consumption. The agent

\[
idle(0.5) \odot P
\]

specifies a behaviour expression that causes time to progress with 0.5 units.

### 3.5.4 Timed Actions

Synchronization actions are timeless. Input, output and state manipulating actions can consume a specific amount of time, now specified by a time expression. E.g. the behaviour expression

\[
\text{input}(data, *te), *tv \odot P
\]

specifies a prefix construction with an input action *input* which during communication receives a value and binds it to *data*. The time it spends waiting on the environment is recorded in time variable *at*. The actual time of communicating the input value is preset to take *te* time units. The main difference between the time expression *te* and the time variable binding *tv* is the application nature. *expression* specifies a certain amount of time, while *identifier* specifies a value binder receives a value by actions within the system. The time expression is similar to the output actions and function calls, the time variable bindings is similar to input actions and function definitions.

In many cases *te* will be neglectable with respect to the value of *tv*, i.e. *te \ll tv* and may be omitted. But when the size of the value communication begins to grow, the value of *te* may become a factor of importance, and then it may be included.
3.5. Time specification

3.5.5 Examples

Example 3.10 (Time passing through agent value part) An agent definition may contain a value-part specification (Type I agents), that may be regarded as a formal parameter type specification similar to the case of function specifications. The actual value for the formal parameter is obtained at the invocation of an agent service construction. This value is bound by pattern-matching against the patterns within the behaviour definitions to identifiers that make the value applicable within the body of the behaviour definitions. A general form to record time consumption, and passing it to an agent is as follows

$$\text{act}, *t \otimes P (t)$$

where the type of the value-part of agent $P$ is defined as $T$. Time progression starting from the moment that action $\text{act}$ is available to be taken until the moment in time that $\text{act}$ is taken is registered in time-variable $t$ and upon the moment of the $\text{act}$ transition substituted for the formal occurrences of $t$ within the value-part of $P$ only.

Example 3.11 (Stopwatch) A particular model of a stopwatch consists of two buttons for control and one for reading the value of the stopwatch. There is a start and stop button. Reading can be done only after stopping the stopwatch. Specification 3.18 presents a MOSCA model. After the stop-

<table>
<thead>
<tr>
<th>3.18 Stopwatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SW$</td>
</tr>
<tr>
<td>ports syn start</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$SW \triangleq$</td>
</tr>
</tbody>
</table>

watch is started, the agent $SW$ is ready to accept stop. The time elapsed between start and stop is registered in time variable $t$.

A more realistic usage of time recording is presented in the next example. Notice the usage of an unit, to enable the usage of an internal value part ($T$) to record the time progression.

Example 3.12 (Another Stopwatch) A more realistic stopwatch can be started and stopped with one button, reset with another and constantly read. Its ticking value is e.g. preset to 1 centisecond.\(^2\) Its specification is given in spec. 3.19.

\(^2\)Assuming a second as the unit in which the progression of time is recorded.
unit \textit{RSW} \\
ports syn \textit{ss} \\
\hspace{1em} \textbf{syn} \ \textbf{reset} \\
\hspace{1em} \textbf{out} \ \textbf{display} : T \\
shares \textit{RSWR} \langle T \rangle \\
\hspace{1em} \textit{RSWS} \langle T \rangle \\
\textit{RSW} \triangleq \textit{ss} \odot \textit{RSWR} \langle 0 \rangle \\
\textit{RSWR} \langle t \rangle \triangleq \\
\hspace{1em} (\text{idle}(\text{tick}) \odot \textit{RSWR} \langle \text{tick} + t \rangle) \oplus \\
\hspace{1em} \textit{ss}, \star d \odot \text{idle}(\text{tick} - d) \odot \textit{RSWS} \langle \text{tick} + t \rangle \oplus \\
\hspace{1em} \text{display}(t), \star d \odot \text{idle}(\text{tick} - d) \odot \textit{RSWR} \langle \text{tick} + t \rangle \\
\textit{RSWS} \langle t \rangle \triangleq \textit{ss} \odot \textit{RSWR} \langle t \rangle \oplus \textbf{reset} \odot \textit{RSW} \\
where \\
values \\
\textbf{tick} : T = 0.01 \\
end

The stopwatch can be in three states, ready for a fresh start — \textit{RSW}, ticking — \textit{RSWR} and ready to be restarted after a previous ticking status, retaining the current reading — \textit{RSWS}. In the expression

\[(\text{idle}(\text{tick}) \odot \textit{RSWR} \langle \text{tick} + t \rangle) \oplus \textit{ss}, \star d \odot \text{idle}(\text{tick} - d) \odot \textit{RSWS} \langle \text{tick} + t \rangle\]

either the idle action runs to completion, whereafter the first prefix will be active, or within the timespan of a tick the start/stop button is pressed, after which the second prefix will be activated.

Let's continue our series of buffer specifications with two versions of buffer that both apply a notion of time.

\textbf{Example 3.13 (Timed Buffer)} On an abstract level a timed buffer could be specified that directly measures the progression of time by time variables. Its definition is presented in specification 3.20.

The function of the timed buffer can also be accomplished using the stopwatch. In specification 3.21 the time registration is encapsulated by the stopwatch agent. Through its port \textbf{display} the actual reading of the time is recorded and registered in the buffer. Within the buffer itself no time variables are applied.
3.5. *Time specification*

\[ \text{unit } TBuffer1 \]
\[
\begin{align*}
\text{ports } & \text{in } \text{in} : \mathbb{N} \\
& \text{out } \text{out} : \mathbb{N} \times \mathbb{T} \\
& \text{syn } \text{start} \\
& \text{syn } \text{stop} \\
\text{shares } & RBuffer \langle (\mathbb{N} \times \mathbb{T})^* \times \mathbb{T} \rangle
\end{align*}
\]

\[ TBuffer \triangleq \text{start } \odot RBuffer \langle [\cdot], 0 \rangle \]

\[ RBuffer \langle (s, tval) \rangle \triangleq \]
\[
\begin{align*}
\text{if } s &= [] \\
\text{then } & \text{in}(v), \star t \odot RBuffer \langle \langle [v, t + tval], t + tval \rangle \rangle \oplus \\
& \text{stop } \odot TBuffer \\
\text{else } & \text{in}(v), \star t \odot RBuffer \langle \langle s \mapsto [v, t + tval], t + tval \rangle \rangle \oplus \\
& \text{out}(\text{hd } s), \star t \odot Rbuffer \langle \langle \text{tl } s, t + tval \rangle \rangle \oplus \\
& \text{stop } \odot TBuffer
\end{align*}
\]

**Example 3.14 (Timing Buffer with Stopwatch)** To enable time registration, the buffer is extended to hold a realistic stopwatch (spec. 3.21). The buffer accepts natural numbers and stores the numbers together with the moment of time, relative to the start of the stopwatch, the entry was added to the buffer. Its activation proceeds via synchronization by the environment through the \textit{start} action. The stopwatch is started and the buffer is ready to accept natural numbers.

The model for the buffer is contained in an unit. The agent \textit{TBuffer} delivers the composition of \textit{RBuffer} with the stopwatch \textit{RSW}. This combination models the active buffer. It is able to accept new values and can deliver buffered values. It can be stopped, after which the behaviour of the \textit{RBuffer} part of the composition continues as \textit{SBuffer}, which stops the stopwatch, resets it, waits for a restart of the buffer, and restarts the stopwatch.

\[ \square \]

Again mark the specific usage of the value part specifications of the execution states. The unit \textit{TBuffer} itself does not offer a value part to the environment, thus effectively hiding the type of the elements within the buffer. The composition of the running buffer \textit{RBuffer} with the stopwatch \textit{RSW} is restricted from any outside interference of the stopwatch through the restriction operator. One clear disadvantage of blocking the availability of ports in this way is the necessary knowledge to the specifier of the visible ports of the agent \textit{RSW} when writing down the restriction. \textit{MOSCA} does not offer a construction that hides all visible ports, without naming them.
unit $TBuffer_2$

ports in $in$ : $\mathbb{N}$

out $out$ : $\mathbb{N} \times \mathbb{T}$

syn $start$

syn $stop$

shares $RBuffer \langle (\mathbb{N} \times \mathbb{T})^* \rangle$

ports in $display$ : $\mathbb{T}$,

$SBuffer$

ports syn $ss$

syn $reset$

$TBuffer \triangleq$

$\text{start} \odot (ss \odot RBuffer \langle [] \mid RSW \rangle \setminus \{\text{display}, ss, \text{reset}\})$

$RBuffer \langle s \rangle \triangleq$

if $s = []$

then $\text{in}(v) \odot \text{display}(t) \odot RBuffer \langle [(v, t)] \rangle \oplus$

$\text{stop} \odot SBuffer$

else $\text{in}(v) \odot \text{display}(t) \odot RBuffer \langle s^{\leftarrow} [(v, t)] \rangle \oplus$

$\text{out}(hd s) \odot Rbuffer \langle tl s \rangle \oplus$

$\text{stop} \odot SBuffer$

$SBuffer \triangleq ss \odot -- \text{stop the stopwatch}$

reset $\odot -- \text{reset the stopwatch}$

start $\odot -- \text{waiting for a restart action for the buffer}$

ss $\odot -- \text{start the stopwatch}$

$RBuffer \langle [] \rangle$
3.6 Summary

MOSCA can be briefly characterized as the union of four groups of constructions: VDM-SL, CCS basic behaviour constructions, loose behaviour constructions and time description.

\[
\text{MOSCA} = \text{VDM-SL} + \text{BBC} + \text{LBC} + \text{TIME}
\]

The MOSCA notation takes the VDM-SL language as value manipulation language. Processes are specified by an agent definition. The language supports agent definitions in a functional and an imperative style. On top of the agent definition mechanism MOSCA offers a simple structuring ability allowing the definition of collections of entities which can be exported and imported into specifications.

The set of basic behaviour constructions (BBC) contain the basic CCS operations like prefixing, choice, composition, restriction and relabelling is enriched with loose behaviour constructors (LBC) like agent service, agent if, and agent let be. Further the notation is extended with timed actions and the delay operator (TIME).

The basic behaviour operators are restricted to a small set in comparison to other notations like CSP and LOTOS. CSP has a rather rich set of operators. It provides operators for prefix, non-deterministic and deterministic internal choice and external choice, hiding and relabelling, interleaving, chaining, subordination, interrupting, restarting, alternating, repeated and conditional behaviour (see [103] for details). LOTOS has some other specific constructions like explicit interleaving, enabling and disabling. Other approaches offer in addition to the action prefix construction an action sequence construction \(';\). The sequencing operator gives the notation a more imperative character.

The time description facilities has been kept simple. Only timed actions and the delay construction are provided. I feel that these two constructions are sufficiently strong to model the current paradigms with respect to time handling. A major design decision has been the choice between including single-point time specification and interval specifications. The semantics of both forms can be expressed in a SOS setting, but the single-point approach leads to a more operational setting than the interval approach. The interval approach, applied by for example CHEN+ and DANIELS does not offer a kind of failure semantics. If the action is not taken within the specified time interval the whole system is meaningless.
Open intervals like 

\[ a@[l, \infty]. P \]

are easily modelled in MOSCA's single-point approach:

\[ \text{idle} l \odot a, \ast t \odot P \]

Closed intervals resemble more intriguing modelling. In a behaviour expression like 

\[ a@[l, h]. P \]

the action \( a \) should be taken after \( l \) time units but before \( h \) time units. Whenever time progresses beyond the upper limit of the interval the action cannot be taken any more and the time, and hence the behaviour is frozen. In the single-point approach, the above behaviour can be simulated by 

\[ \text{idle} l \odot a, \ast t \odot P \odot \text{idle} h. \bot \]

where the behaviour expression \( \bot \) denotes the diverging process, forever occupied with internal actions. Another approach to model closed intervals can be found in the application of the conditional behaviour expression. E.g.

\[ \text{idle}(t0) \odot a, \ast t \odot \text{if } t \leq t1 \]

\[ \begin{align*}
\text{then } & \text{Bexpr1} \\
\text{else } & \text{Bexpr2}
\end{align*} \]

forces the behaviour \( \text{Bexpr1} \) to be selected whenever the action \( a \) is taken within the closed interval \([t0, t1]\), and activates \( \text{Bexpr2} \) whenever \( a \) is taken after \( t1 \) units of time, if ever. This approach inhibits time stops, and subsequently system stops, as is the case with CHEN's approach. To signal inappropriate system behaviour \( \text{Bexpr2} \) can be set equal to null at early specification phases and be specified later as exceptional behaviour.

Summarizing MOSCA, we get the schematic setup presented in figure 3.6.
Figure 3.6: Mosca features summary
Semantics

In this chapter I study the semantic basis of MOSCA. The main contribution of the research that resulted in the MOSCA semantics is the definition of an operational semantics based on the SOS approach in which looseness is integrated without any artificial extensions of the basic SOS structure. Looseness is treated as another source of non-determinism. Section 4.1 presents the basic idea of loose process behaviour semantics, by studying the semantics of a small language called LOOP.

In section 4.2 the choices are presented through which the semantics of MOSCA became assembled. Section 4.3 opens the semantic definition of MOSCA with a summary of the dynamic semantics of VDM-SL. In section 4.4 a combination semantics is defined, based on a combination of an extended form of a labelled transition system to describe process behaviour and time and a relational denotational semantics to denote models for value constructions. The states in the transition system are formed of a syntactic component, the behaviour expression that characterises the continuation behaviour of the state and a semantic component, the environment that holds all bindings from identifiers to value and agent denotations. The section discusses the structure of the environment, the action set (i.e. the labels) of the transition system, the transition relation and the connection of the transition semantics to the denotational semantics.

In section 4.6 an important property of MOSCA process definitions is studied, the property of loose specification.

4.1 Loose process semantics

In this section I will present a semantics for a small language with looseness. The overall setup of the semantics is similar to the setup of the semantics given for MOSCA. The language offers both behaviour expressions and value expressions. The behaviour expressions are composed of the prefix, choice
and composition operator. The prefix is restricted to input and output prefixes. The value expressions are constructed from constants, addition and a loose construction which is a simple form of the value-let-be construction of VDM-SL. Assume the following abstract syntactic domains.

\[
\begin{align*}
\pi \in \Pi & : \quad BehaviourExpressions \\
\gamma \in \Gamma & : \quad ProcessConstants \\
\lambda \in \Lambda & : \quad PortLabels \\
\xi \in \Xi & : \quad ValuePatterns \\
e \in E & : \quad ValueExpressions \\
\nu \in N & : \quad Numbers
\end{align*}
\]

The syntactic domain *Numbers* contains representations for a large enough set of number constants, for example the natural numbers. The language LOOP, an acronym for Loose Processes, consists of processes defined over the following abstract syntax.

\[
\begin{align*}
\Pi ::= & \quad \pi|\pi \quad (composition) \mid \\
& \quad \pi + \pi \quad (choice) \mid \\
& \quad \lambda?\xi.\pi \quad (input \ prefix) \mid \\
& \quad \lambda!e.\pi \quad (output \ prefix) \mid \\
& \quad \gamma \quad (constant \ instantiation) \\
E ::= & \quad \nu \quad (value \ constant) \mid \\
& \quad \xi \quad (value \ pattern) \mid \\
& \quad e + e \quad (value \ addition) \mid \\
\text{let } \xi \in N \text{-set in } E \quad (loose \ let \ binding)
\end{align*}
\]

In the let expression *N*-set forms the basis over which the value pattern \( \xi \) may range. To avoid unnecessary complication I restrict the set elements from being specified in a loose manner. A concrete syntax example for a let expression could be something like

\[
\text{let } x \in \{1, 2, 3\} \text{ in } x
\]

resulting in an expression with three different possible values. The denotation of a value expression is an element of the following semantic domain:

\[
v \in \text{VAL} : \quad Valuedenotations
\]
which contains denotations for the elements in the syntactic domain *Numbers*. Further, let

\[ \rho \in ENV: \ \text{Environments} \]

denote a recording of bindings between process constants and behaviour expressions and value patterns and value denotations, such that:

\[ ENV :: venv : VENV \\
penv : PENV \]

where

\[ PENV = \Gamma \xrightarrow{m} \Pi \]

\[ VENV = \Xi \xrightarrow{m} VAL \]

The semantic function `eval` computes the value denotations. It is given in specification 4.1. I have omitted the details with respect to the three auxiliary functions `ComputeNumberDenotation`, `SemPlus` which are straightforward and `ConvertToSet`, which depends on the syntactic representation of the set. The evaluation of `eval(e)` results in a set of functions from environments to value denotations. The addition is taken into the language for the sake of its semantics. It is a typical case of an operator that propagates looseness. The semantic result of `eval(a + b)` consists of a set of functions where each function combines the effect of looseness of `a` with the effect of
looseness of $b$ by selecting one particular value for both arguments. Suppose that the cardinality of the set of semantics values for resp. $a$ is $\mathcal{N}(a)$ and $b$ is $\mathcal{N}(b)$, then the cardinality of the set of values for $eval(a + b)$ is $\mathcal{N}(a) + \mathcal{N}(b)$. In the let expression the cardinality of the set of semantic values for the whole expression is fixed by the number of semantic values for the body of the expression $e$ multiplied by the cardinality of the base set $s$.

Let $\pi$ range over $\Pi$ and $\rho$ range over $ENV$. Let further $\lambda \in \Lambda$, $\nu \in N$ and $D = \{?,!\}$. The semantics for LOOP is defined by a labelled transition system

$$\langle \text{State}, \text{Action}, \rightarrow \rangle$$

where $\text{State} = \Pi \times ENV$ and $\text{Action} = (\Lambda \times D \times VAL) \cup \{\tau\}$, and $\rightarrow$ is a relation over

$$(\Pi \times ENV) \times \text{Action} \times (\Pi \times ENV)$$

specified as usual as the least relation defined by a collection of inference rules. These rules are formatted in a form similar to the standard SOS rules given in section 2.3.3. The hypothesis and conclusion are placed in a box labelled ‘H’ and ‘C’ respectively.

Rule 4.1 Input Prefix

<table>
<thead>
<tr>
<th>rule Prefix-in</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Here the empty hypothesis box reflects the fact that the inference can always take place. The first two lines in the conclusion construct a new environment in which the new value binding from $\xi$ to $v$ is recorded. Notice that in the third line the action specifies both the port name $\lambda$ and the semantic value $v$ that resulted from an associated output action. The inference rule for the output prefix is presented next.

Rule 4.2 Output Prefix

<table>
<thead>
<tr>
<th>rule Prefix-out</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Again there is no hypothesis. In the conclusion the first two lines compute a set of value denotations for the output expression \( e \). Then for each value denotation there is in fact a separate rule that provides the output action for that specific value. These rules are merged in notation into one rule, by use of the universal quantifier. As such the looseness resulting from the evaluation of the output expression is transformed into a non-deterministic behaviour of the prefix expression containing the loose output expression.

For the composition and choice the standard rules apply. The rule for composition that describes the effect of synchronization is given next.

**Rule 4.3 Composition with synchronization**

<table>
<thead>
<tr>
<th>rule Composition-s</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 ( (p, \rho) \xrightarrow{(\lambda? v)} (p', \rho_p) ) ( (q, \rho) \xrightarrow{(\lambda! v)} (q', \rho) )</td>
</tr>
<tr>
<td>C 2 ( (p \parallel q, \rho) \xrightarrow{\tau} (p' \parallel q', \rho_p) )</td>
</tr>
</tbody>
</table>

As usual I let \( \tau \) denote the action label that signals synchronization. The rule for constant substitution reflects the extend of the value bindings.

**Rule 4.4 Constant Unfolding**

<table>
<thead>
<tr>
<th>rule Constant Unfolding</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 ( (p, ({ }, penv)) \xrightarrow{a} (p', \rho') ) ( penv(q) = p )</td>
</tr>
<tr>
<td>C 2 ( (q, (venv, penv)) \xrightarrow{a} (p', \rho') )</td>
</tr>
</tbody>
</table>

I have chosen to restrict the extend of the value bindings to the first application of a process constant unfolding.\(^1\) The effect of the transition steps can be recorded into a derivation tree. Suppose we start with the state

\[ ((\lambda? \xi. p1) \parallel (\lambda! \text{let } x \in \{1, 2, 3\} \text{ in } z.p2), \rho). \]

According to the inference rules above the state has three different transition steps, which results in the following partial derivation tree.

\(^1\)The same approach is followed in the MOSCA semantics.
The looseness in the value expression results in three possible different transition paths. The behaviour is non-deterministic in the sense that the semantics does not preselects a specific transition path. All three transitions are candidates, but only one will actually happen.

The language LOOP is only meant as a demonstration vehicle. The overall strategy to define the semantics is the same as the strategy taken to fix the semantics for MOSCA. The overall semantics is operational and contains the denotational value semantics as embedded semantical structure. Figure 4.1 presents the overall structure of the semantics of LOOP.

4.2 Overview of the semantics of MOSCA

To define the semantics of MOSCA, starting off with VDM-SL and CCS, choices had to be made from several possible semantic models on which MOSCA could be based. The meaning of VDM-SL specifications is currently being fixed by a denotational semantics within a relational setting in [136]. CCS had been given a whole range of semantics, starting from structural operational semantics in [94], [157], through equational theories in [155], [100], to full denotational semantics in [99], [98]. Time description in combination with process description through CCS is widely studied lately. The comparison of the time extensions to CCS, has resulted in the current time description facilities in MOSCA.

The basis for the value semantics could either be denotational, like as is currently undertaken for VDM-SL, or structured operational. Pragmatic reasons have led me to encompass the relational denotational semantics of VDM-SL. It is not a trivial task to setup a structured operational semantics
4.2. Overview of the semantics of MOSCA

Figure 4.1: Semantic Structure of LOOP
for VDM-SL from scratch, the mere bulk of the language would ask for a substantial investment of man-power.

The basis for the process semantics of CCS could also be either denotational or structured operational. A denotational model would probably fit nicely to the denotational semantics of VDM-SL. This issue is not studied here, although. As a result from my investigations related to the incorporation of time, it appeared that all time extensions of CCS start from the structured operational approach through labelled transition systems. Throughout the process of finding an appropriate semantical basis for MOSCA, I have pursued the philosophy of combining and extending existing approaches.

The outcome has resulted in the adaptation of the structured operational model, extended to cater for the time and value extensions. The transition system used for fixing the meaning of behaviour expressions is a labelled transition system, lts for short, extended to capture time and a notion of an environment. The behavioural aspect of behaviour expressions follows from the state transitions in the extended lts that are initiated from the expression.

A summary of the semantic structure of MOSCA is presented in figure 4.2. The starting point for the semantics is a representation of a MOSCA specification in core abstract syntactic form (CAS), which is build from both agent constructs and value constructs. The agent constructs are assembled into an environment, together with the value denotations for the value constructs. The initial behaviour expression, together with the environment form the basic sts on which the structured operational semantics is defined. The semantic function VS first transforms the abstract value constructs to similar constructs within the VDM-SL CAS representation, followed by applying the specific VDM-SL semantic mapping to achieve appropriate value denotations. In the following sections each of the constituents of the MOSCA semantics is explained.

4.3 VDM-SL Semantics

Starting point for the current form of definition of the semantics of VDM-SL has been [14], a preliminary definition that resulted from a project under supervision of Dines Bjørner and Peter Mosses. The project was based on earlier work from MONAHAN in [160], [161] and BEAR in [24]. The work has been presented at the IFIP world congress 1989 ([138]). Early 1989 a first complete draft version of the semantics was finished and circulated
Figure 4.2: Semantic Structure of MOSCA
within a Review Board installed by VDM-EUROPE\textsuperscript{2} The outcome of the review process was manifold. One of the most important changes was the adoption of a more dedicated domain universe, proposed by TARLECKI in [205], worked out by Wieslaw Pawlowski and Peter Larsen. At the first ISO/VDM meeting in April 1991 a new review board was installed chaired by Kees Middelburg. As being a member of this review board, I have had the opportunity to study the semantics of VDM-SL in detail. The main reference to the dynamic semantics will be the future ISO VDM-SL standard. Although the dynamic semantics are, at the moment of writing, relatively stable, it is still possible that changes in the domain definitions and semantic functions will be made. The reference for the work recorded in this thesis is [108].

In this section a brief summery of the semantic approach with respect to value denotations is sketched. First the basic semantic domains, based on the notion of cpo's are shortly introduced. Next the overall strategy is explained.

4.3.1 The universe of Complete Partial Orders

The mathematical model on which the semantic domains for VDM-SL are built is the well defined notion of a complete partial ordering (cpo). The carrier sets are based on the VDM-SL basic value sets of boolean values, characters, the nil value, the token values, the numbers, ranging from natural numbers, to real numbers and the quotation values (see [108]). These value sets are transformed into basic tagged cpo's. In a tagged cpo each value of a carrier is tagged, i.e. enlarged to a tuple, in which a carrier tag (type tag) is injected. These tagged cpo's form the basis for the construction of an universe of cpo's as described by TARLECKI in [205]. A brief summery of the construction of the universe of cpo's is as follows. The construction starts with the family of basic cpo's and proceeds by closing it under the basic cpo operators which are the subset operator, the cartesian product operator, the finite sequence operator, the record space, mapping space, and function space operators. Each added cpo is tagged with the appropriate tag. Then the unions of all countable many union-compatible sets of cpo's in the family are added. Unfortunately, since not all of the basic operators preserve the unions added, the family of cpo's obtained is not closed under the operators. Hence, the procedure is iterated over, bound by the first uncountable ordinal $\omega_1$. The unions of all families in the hierarchy of cpo's is then taken to be the basis for the value domain of

\textsuperscript{2}Now FME, Formal Methods Europe, an advisory board to the CEC, aimed at the widespread of use of formal methods in industry.
4.3. VDM-SL Semantics

VDM-SL, CPO.

It has been proved by TARLECKI in [205] that the cpo universe CPO thus created contains the basic cpo’s, a least element, is closed under all basic cpo operators, as well as under unions of countable many union-compatible sets of cpo’s.

4.3.2 Basic Semantic Domains

The basic semantic domains are given in spec. 4.2.

- The top domain reflects the flat textual structure of VDM-SL. A value environment consists of a mapping from identifiers to denotations. All identifiers are unique.

- VAL is the domain of all possible value denotations, i.e. an element of the domain VAL belongs to a carrier set and that carrier set is in the domain DOM. The domain VAL has no cpo structure. It consists of all carrier sets of the cpo’s in the domain DOM.

- The domain DOM contains denotations for all VDM-SL types. An element \( (cpo, invs) \in DOM \) consists of a cpo and a set of elements from the carrier of the cpo. The set invs reflects a possible invariant on the VDM-SL domain.

- The domain OPVAL contains all VDM-SL operation denotations. A value from this domain is a set of values \( (in, ost, nst, out, mode) \) where \( in \) is a denotation of the input argument of the operation, \( out \) a value denotation, possible nil for the output value of the operation, \( ost \) and \( nst \) denotations resp. for the state value before and after the operation. Operations are interpreted as relations.

- A FLATVAL value denotation is a value \( v \in FLATDOM \) for which the flatness criterion holds, i.e. two denotations are ordered if one of them is the bottom value \( \bot \), or they are equal. In this way the function spaces are ruled out, as they represent non-flat values.

- The domain POLYVAL denotes all polymorphic functions. These denotations are represented as functions from a list of domains to values. The argument domains correspond to an actual instantiation of the function’s type variables.

- Finally, POLYDOM represents all domains for the polymorphic functions.
4.2 Basic Semantic Domains

\[
\begin{align*}
VENV &= Id \xrightarrow{m} (VAL | DOM | OPVAL | POLYVAL | LOCVENV) \\
VAL &= CARRIER \\
\text{inv } v &\triangleq \exists (\text{cpo, invs}) : DOM \cdot \text{let } mk-(c, o) = \text{cpo }\text{ in } v \in c \\
DOM &= CPO \times CARRIER-set \\
\text{inv } mk-(mk-(\text{carrier-s, -}), \text{invs}) &\triangleq \\
\forall \text{ival }\in \text{invs} \cdot \text{ival} \neq \bot \land \text{ival }\in \text{carrier-s} \\
OPVAL &= FLATDOM \times FLATDOM \times FLATDOM \times (FLATDOM | \text{nil}) \times MODE-set \\
MODE &= \text{exit} \mid \text{cont} \\
FLATDOM &= DOM \\
\text{inv } mk-(mk-\text{carrier-s, o), inv} &\triangleq \\
\forall a_1, a_2 \in \text{carrier-s} \cdot o(a_1, a_2) \iff a_1 = \bot \lor a_1 = a_2 \\\nFLATVAL &= CARRIER \\
\text{inv } v &\triangleq \exists (\text{cpo, inv}) : FLATDOM \cdot \text{let } mk-(c, o) = \text{cpo }\text{ in } v \in c \\
POLYVAL &= DOM^+ \rightarrow VAL \\
POLYDOM &= DOM^+ \rightarrow DOM
\end{align*}
\]
4.3. VDM-SL Semantics

4.3.3 Overall Strategy

The value semantics are defined upon a core abstract syntax, from which the top node is the Definitions structure (see appendix B for the whole core abstract syntax for MOSCA).

\[
\text{Definitions} ::= \quad \text{typem} : \text{Id} \rightarrow \text{TypeDef} \\
\text{expofnm} : \text{Id} \rightarrow \text{ExplPolyFnDef} \\
\text{ezmofnm} : \text{Id} \rightarrow \text{ExplFnDef} \\
\text{impo_fnm} : \text{Id} \rightarrow \text{ImplPolyFnDef} \\
\text{immo_fnm} : \text{Id} \rightarrow \text{ImplFnDef} \\
\text{valuem} : \text{Id} \rightarrow \text{ValDefs} \\
\text{explotm} : \text{Id} \rightarrow \text{ExplopDef} \\
\text{implplotm} : \text{Id} \rightarrow \text{ImplOpDef} \\
\text{state} : \text{[StateDef]}
\]

The semantics of a value specification written in VDM-SL is given by the SemSpec function. The structure of a value specification is simply a collection of CAS representations of definitions. The semantics of such a collection of definitions is the (possible infinite) set of models fulfilling all the definitions. A model will fulfill a collection of definitions if it provides appropriate denotations to all the definitions. Thus, a model can also be considered as the environment which the collection of definitions provides for the 'outside' world. The semantics of a value specification is a set of models because the definitions may be loosely specified.

The SemSpec function is given in an implicit style. Its definition is in specification 4.3, and its main auxiliary function, IsAModelOf is in specification 4.4. The function SemSpec tests all elements of the domain VENV

\[
\begin{align*}
\text{SemSpec} : \text{Definitions} & \rightarrow \text{VENN-set} \\
\text{SemSpec}(\text{defs}) & \triangleq \{\text{env} \in \text{VENN} \mid \text{IsAModelOf(env, defs)}\}
\end{align*}
\]

against the IsAModelOf predicate. This style deviates from the normal explicit style of semantic definitions, which is traditionally used to define the meaning of programs. STOY gives in [202] a traditional account on the theory of denotational semantics. In [203] the setting is related to VDM-SL forerunner Meta-IV. Other references to presentations of the theory of denotational semantics are e.g. [9], [196] and [206]. In order to give a compositional semantics the explicit style requires an order in which the definitions must occur. The VDM-SL does not have this property. The implicit style, also marked the relational style originates from [160] where it was used to define the semantics of STC VDM.
The `IsAModelOf` predicate shows how a syntactic specification is related to an already given candidate model. The `Verify` predicates check whether or not the given model is really a candidate model for the syntactic specification. E.g. the first verification predicate `VerifyTypes` is defined as follows.

\[
\text{VerifyTypes} : (\text{Id} \xrightarrow{n} \text{Typedef}) \times \text{VENV} \rightarrow \text{B}
\]

\[
\text{VerifyTypes}(\text{tpdef}_m, \text{env}) \triangleq \\
\text{if } \text{dom } \text{tpdef}_m \subseteq \text{dom } \text{env} \land \forall \text{id} \in \text{dom } \text{tpdef}_m \cdot \text{env}(\text{id}) \in \text{DOM} \\
\text{then let } \text{env}' = \text{EvalTypeDef}(\text{tpdef}_m)(\text{env} \setminus \text{dom } \text{tpdef}_m) \text{ in} \\
\quad \text{if } \text{env}' = \text{err} \\
\quad \text{then false} \\
\quad \text{else } \forall \text{id} \in \text{dom } \text{tpdef}_m \cdot \text{env}(\text{id}) = \text{env}'(\text{id}) \\
\text{else false}
\]

First it is checked that all names of the type definitions belong to the domain of the environment, and that they are bound to proper type denotations. Then a new environment `env'` is constructed by evaluating the syntactic type definitions in the given environment from which the types are extracted. If the computed denotations for the types (here each type has exactly one denotation, as types may not be specified loosely) match the denotations in the given environment only by one the given environment is accepted, otherwise rejected.
4.3.4 The evaluation functions and domains

Most of the Eval- functions return a set of value evaluators resulting from the possible underdeterminatedness of functions, expressions and patterns. Prior to calling the verification predicates, IsAModelOf will expand the given model with constructs which are implicitly defined. The semantic domains for the Eval- functions are defined in specification 4.5. They are given here as illustration to the overall style of the semantic definition. They are used in the definition of the semantics of MOSCA further on. These domains can roughly speaking be divided into two groups, the definers and evaluators. A (possibly loose) definer can be considered as a function that changes the current environment. A (possibly loose) evaluator can be considered as the semantic counterpart of a syntactical entity. The domain Def consists of total functions on environments which may return an error indication. The type evaluators TEval, from which EvalType is an example, return one function that, given an environment, computes the type denotation. The statement evaluators SEval compute, given an environment, a triple. The first component is an environment containing possibly changed state parts. The second component indicates whether evaluation ended normally, or whether an exit was encountered. The third component holds any computed value, like e.g. for return statements. The pattern evaluators in PatEval compute a function from value denotations to functions from environments to new environments enriched with the bindings resulting from the pattern matching. An expression evaluator in EEval results in a function from an environment to a value denotation.

4.4 MOSCA’s transition system

In this section I develop the semantic model on which the meaning of the behaviour expressions is defined. The chosen model starts from Plotkin’s SOS model and exhibits three particular properties, i.e.

- the state domain of the lts is extended to hold a notion called environment, capturing bindings from identifiers to both semantic value denotations and syntactic agent definitions;

- the label domain is extended to hold values from (i) the set of visual ports (as usual), (ii) values of the semantic domain VAL and (iii) values of the semantic domain TIME;

- the transition relation reflects the looseness that is an inherent quality of the value specifications of MOSCA. To this end the transition rules are defined by quantification over sets of states.
4.5 Evaluation Function Domains

\[ LDef = \text{Def-set} \]

\[ \text{Def} = VENV \rightarrow (VENV | \text{err}) \]

\[ T\text{Eval} = VENV \rightarrow (\text{DOM} | \text{err}) \]

\[ LSE\text{val} = SE\text{val-set} \]

\[ SE\text{val} = VENV \rightarrow ((VENV \times \text{MODE} \times (\text{VAL@und} | \text{nil})) | \text{err}) \]

\[ LP\text{atEval} = \text{PatEval-set} \]

\[ \text{PatEval} = \text{VAL} \rightarrow \text{BindEval} \]

\[ \text{BindEvals} = \text{Bindval-set} \]

\[ \text{BindEval} = VENV \rightarrow (\text{VEnv-set} | \text{err}) \]

\[ V\text{Env} = \text{Id} \xrightarrow{m} \text{VAL@und} \]

\[ LEE\text{val} = \text{EEval-set} \]

\[ \text{EEval} = VENV \rightarrow \text{VAL@und} \]

\[ LP\text{E}eval = \text{PEval-set} \]

\[ \text{PEval} = VENV \rightarrow \text{POLYVAL@und} \]

\[ P\text{DomEval} = VENV \rightarrow \text{POLYDOM} \]
As a result both the state space and the set of transition rules become (wildly) infinite. Although the infinity aspect has difficult theoretical properties (e.g. see [104] for an analysis concerning TCCS) the lts remains a solid base for the operational description of the behaviour of agents.

4.4.1 The environment

To define the behaviour semantics of MOSCA the notion lts is extended to include an environment in the following sense.

Definition 4.1 (elts) An extended labelled transition system elts is a triple

\[((P \times R), \text{Act}, \rightarrow)\]

where \(P\) ranges over behaviour expressions, \(R\) ranges over environment expressions and \(\rightarrow\) is extended to a relation

\[(P \times R) \times \text{Act} \times (P \times R)\]

An environment \(\rho\) captures the set of definitions in which processes execute. These definitions include all data definitions like types, values and functions, agent definitions, the set of local bindings resulting from value let constructions, pattern matching etc. Environments are associated with processes. In a composition of processes \(A \mid B\), each process retains its own environment. The environment expression \(R\) is created from environments \(\rho\), using environment composition denoted by the \(|\rangle\langle|\) symbol and environment choice, denoted by ‘+’. The structure of the environment expression is equal to the structure of the behaviour expression. E.g. the behaviour expression \((E1 \mid (E2 \mid E3))\) will be associated with the environment expression \((\rho_1 \mid (\rho_2 \mid \rho_3))\) where \(Ei\), denote arbitrary behaviour expressions and \(\rho_i\) their private environment. The environment choice operator ‘+’ enters the environment as a result of the idling as explained in section 4.5. The following definition introduces the tuple space for \((P, R)\) precisely.

Definition 4.2 (sts) A state transition state, or short sts, is an element of the domain

\[STS = (\text{Spec} \mid \text{Bexpr}) \times \text{Eexpr}\]

\[\text{inv}(\text{bexpr}, \text{eexpr}) \triangleq \text{Equal\_Structure}(\text{bexpr}, \text{eexpr})\]
such that the composition and choice structure of the environments matches
the composition and choice structure of the behaviour expressions. The
states within the els are constructed as tuples $(P, \rho)$, where $P$ ranges over
the domain $Spec \mid BExpr$. The start of a system as a whole is through the
$Spec$ construction. Each $Spec$ defines a set of definitions and a behaviour
expression. $BExpr$ is the domain of behaviour expressions for agents. The
second element of the tuple, $\rho$, is an environment expression.

**Definition 4.3 (Environment expression)** An environment expression
ranges over the following domain.

$EExpr = Environment \mid EExpr || EExpr \mid EExpr + EExpr$

An environment is a tuple consisting of five parts, defined by the next
domain definition:

$Environment :: Gvenv : VENV$
$Cvenv : VENV$
$Stenv : VENV$
$Gae : AENV$
$Cae : AENV$
$cae : [Id]$

The different constituents of the environment have the following function:

- $Gvenv$ and $Cvenv$ range over the value domain $VENV$ defined by the
top level semantic domain for the value denotations (see spec. 4.2). $Gvenv$
collects all global value part definitions, $Cvenv$ acts as actual (current)
value environment and adds all local value bindings to the
global value environment. The value environment reflects the the flat
scope structure of MOSCA specifications: only level for global entities
and one level for current entities. The current value environment rec-
ords all bindings that result from value parts, in actions, agent let
constructions and time variables of timed prefixes. The global envi-
ronment is distributed to all constituents of environment expressions,
thus realizing a total distributed name-space without any shared data.

- $Stenv$ collects a copy of the semantic equivalent of the values of a
VDM-SL state definition that may appear in agent and units. In an
agent the state can only be manipulated within a single control thread.
In units the state may be manipulated by each execution state. These
execution states can execute in parallel sharing the state, although
completely without any interference. Each state manipulation is restricted syntactically to state-manipulating prefix expressions. Semantically these state manipulations are atomic and the constituents cannot be interleaved. Let \( s_i \) be MOSCA statement, then

\[
\sigma(s_1; s_2) \odot \sigma(s_3) \neq \sigma(s_1) \odot \sigma(s_2; s_3) \neq \sigma(s_1) \odot \sigma(s_2) \odot \sigma(s_3),
\]

that is, \( \odot \) and \( ; \) do not commute. On each state-manipulating prefix the state values recorded in \( Stenv \) are copied into \( Cenv \). Next a new value environment is computed that reflects the effect of the state manipulation, and the state values from the new environment are copied out into \( Stenv \). In a composition the changes are propagated to the environment of the composition itself (see the section on environment manipulation further on).

- \( Gaenv \) and \( Caenv \) range over the syntactic domain of agent definitions.

\[
AENV = Id \xrightarrow{m} AgDef | UnitDef | StrDef | BExpr
\]

\( Gaenv \) collects all global syntactic agent definitions, all syntactic unit definitions, all syntactic stream definitions. \( Caenv \) adds to the global agent environment all bindings between identifiers and behaviour expressions that result from the evaluation of agent let expressions (each referenced by an unique name);

- The last item, \( cae \), records the identifier of the current agent environment. Its information is needed to decide whether state values must be re-initialised at an agent service construction.

As such the environment \( \rho \) has a double nature. It records

1. all possible value bindings, i.e. bindings between identifiers and semantic denotations of value constructions such as types, functions, values, operations and states, and

2. all possible agent bindings, i.e. bindings between identifiers and syntactic representations of agent definitions.

The initial environment is constructed from the core abstract syntax domain \( spec \). The initial \( sts \) is constructed from the top level domain of the core abstract syntax and the empty environment \( nil \). The rules that influence the extend of the bindings in the environment are explained in section 4.4.6.
4.4.2 The action set $Act$

**Definition 4.4 (The action set $Act$)** The action set $Act$ for MOSCA is defined by

\[
\begin{align*}
ExtActs & = \text{InActs} \cup \text{OutActs} \cup \text{SynActs} \\
TimeActs & = \{ \epsilon d \mid d \in TIME \} \\
IntAct & = \iota \\
Act & = ExtActs \cup \overline{ExtActs} \cup TimeActs \cup IntAct
\end{align*}
\]

(4.1)  

(4.2)  

(4.3)  

(4.4)

where the co-name operation — is defined by:

\[
\begin{align*}
\forall s \in \text{SynActs} \cdot \overline{s} & = s \\
\forall i \in \text{InActs} \cdot c?\text{val} & = c!\text{val} \\
\forall o \in \text{OutActs} \cdot c!\text{val} & = c?q\text{val}
\end{align*}
\]

(4.5)

The input actions $\text{InActs}$ are denoted by $c?q\text{val}$ where $c$ is the label of an input port, $\text{val}$ a value denotation of the value domain $VAL$. Such a value is e.g. defined by $\mathcal{V}S[\text{veexpr}]$ where $\mathcal{V}S$ denotes the top level semantic interface function for MOSCA values, $\text{veexpr}$ a syntactical form denoting a value expression.

The output actions $\text{OutActs}$ are denoted by $c!\text{val}$, where $c$ is the label of an output port and $\text{val}$ the denotation of a MOSCA value-expression.

The synchronization actions $\text{SynActs}$ are denoted by $\iota$, where $c$ is the label of a synchronization port.

The class of time actions $\text{TimeActs}$ that contains the idle action specifiers $\epsilon(d)$. Each value of $d$ specifies an unit of time. E.g. the transition

\[
P \xrightarrow{\epsilon(d)} Q
\]

denotes that agent $P$ will become $Q$ after $d$ units of time. The action label $\epsilon(d)$ is pure semantical, and is initiated by the syntactic idle action in a prefix construction.

The last action label is $\iota$, labelling an internal action. In MOSCA processes may perform different internal actions: state manipulations, initiated by the MOSCA state manipulating prefix constructions, environment manipulation and synchronization. The semantic representation of the syntactic $\sigma$ action in a prefix constructions is also an internal action.

There is no syntactic $\tau$ action, like in CCS, that models an internal action on the syntactic level. In MOSCA an internal action with very similar semantics is specifiable by taking e.g. a state manipulation with a dummy
4.4. MOSCA's transition system

statement, or an idle action with delay 0. In CCS the action \( \tau \) is used for two different notions. In the first place it models a handshake (synchronization) between two co-named actions in a composition, and secondly it offers a pre-emptive choice construction, like in (following Milner's notation in [157])

\[
A \overset{\text{def}}{=} a.A + \tau.b.A
\]

where the \( \tau \) action in the right operand of the summation operator signifies a transition to a behaviour expression \( b.A \), which is capable of synchronization on the action \( b \) only, where

\[
A' \overset{\text{def}}{=} a.A' + b.A'
\]

offers a summation capable of synchronization on both \( a \) or \( b \). In the definition of \( A \) the behaviour is nondeterministic by virtue of the usage of the \( \tau \) action. The behaviour of \( A' \) is fully deterministic. Moreover, in CCS it certainly holds that \( A \neq A' \).

These two different notions are separated in MOSCA. The \( \iota \) action is pure semantical, results from synchronization, divergence and environment manipulation. The syntactical internal actions can be used to model pre-emptive behaviour in different ways. E.g.

\[
A \overset{\Delta}{=} a \odot A \oplus \text{idle}(0) \odot b \odot A
\]

models an agent similar to the CCS equivalent. The main difference is the inherent time-related non-determinism within the two expressions. In the latter expression the agent either takes the action \( a \) or \( \text{starts} \) to wait for a synchronization on the action \( b \). The choice is instantaneous. By adding a loose let construction the moment of choice can be left unspecified, as in

\[
A \overset{\Delta}{=} a \odot A \oplus \text{let } t \in T \text{ in } \text{idle}(t) \odot b \odot A
\]

Now the agent may wait until the action \( a \) can be taken, or it may, after an unspecified amount of time, decide to wait for a \( b \) action. This last form resembles the effect of the \( \tau \) action of CCS most closely.

4.4.3 The transition relation

The \( \rightarrow \) relation is again defined implicitly as the least relation defined by a set of inference rules. The general form of the inference rules are more complex than in the structural operational semantics of CCS. This is caused by the state extension with environments and the interplay of the loose value semantics with the agent behaviour expressions. The rules are defined in appendix C.
A transition step is denoted with the notation

\[ \sigma_i \xrightarrow{a} \sigma_j \]

with \( a \in \text{ExtActs} \cup \text{TimeActs} \) or

\[ \sigma_i \Longrightarrow \sigma_j. \]

where \( \Longrightarrow \) is an abbreviation for \( \xrightarrow{\cdot} \). \( \sigma_i \) is called the basis of the transition step, \( a \) the action label and \( \sigma_j \) the result of the transition.

The semantic rules fall into different classes: the environment manipulation rules, the loose action rules, the standard action rules, the rules that involve time manipulation, and the rules for null and divergent agents.

**4.4.3.1 Environment manipulation**

Let \( SE: STS \rightarrow STS\text{-set} \) be a function defined by case analysis over the syntactic component in the \( \text{sts} \) argument. The result of this function is a set of \( \text{sts} \) configurations that forms the basis for the transition steps.

\[
\forall \sigma_i \in SE(\sigma_1). \frac{}{\sigma_1 \Longrightarrow \sigma_i} \quad (4.6)
\]

The environment parts of \( \sigma_i \) represents a specific deterministic value denotation which matches the loose value specification within the syntactic component of \( \sigma_1 \). Then for each element \( \sigma_i \) in the set a generic transition rule is defined.

Typical examples of these rules are the rules for specifications, agent and unit service, agent value let and let be expressions, agent agent let and state manipulation prefix.

**Example 4.1 (environment manipulation)** Rule C.6 defines the semantic of agent value let constructions.

<table>
<thead>
<tr>
<th>rules AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( SE(\text{mk-AgentValueLet}(\text{locals, in}), \rho) \triangleq )</td>
</tr>
<tr>
<td>2. \text{let } env = \rho.Cenv \text{ in}</td>
</tr>
<tr>
<td>3. {if ld(env) = \text{err}</td>
</tr>
<tr>
<td>4. \text{then } (\text{mk-Divergent()}, \rho)</td>
</tr>
<tr>
<td>5. \text{else } (\text{in}, \mu(\rho, Cenv \hookrightarrow \text{AddEnv}(env, ld(env))))</td>
</tr>
<tr>
<td>6.</td>
</tr>
<tr>
<td>\forall (B, \rho') \in SE(\text{mk-AgentValueLet}(\text{locals, in}), \rho).</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>C 7. (\text{mk-AgentValueLet}(\text{locals, in}), \rho) \Longrightarrow (B, \rho')</td>
</tr>
</tbody>
</table>
4.4. MOSCA’s transition system

The diagram consists of two parts. The top part defines the typical semantic function \( SE \) for the agent value \( \text{let} \). The bottom part defines the actual transition rule. The diagram defines a set of transition rules, for each element in the \( \text{sts} \) set that resulted from the application of the \( SE \) a rule is defined.

4.4.3.2 Loose action rules

There are three loose action rules: the rules for agent if expressions, prefixes with in actions and prefixes with out actions. In the first case the looseness in introduced by the boolean test expression of the agent if. The looseness in in-actions is introduced by loose pattern-matching of the single (i.e. non-loose) value that is associated with the action label. The out-action looseness results from the evaluation of the expression in an out action.

Example 4.2 The rule for the agent-if expression is defined as follows.

\[
\text{rules Alf}
\]

\[
\begin{array}{c|c}
1 & INF(mk-AgentIf(be, P, Q), \rho) \triangleq \\
2 & \text{let } env = \rho.C env \text{ in} \\
3 & \text{let } testval-s = VS[be](env) \text{ in} \\
4 & \text{let } res-s = \{(\text{cases } tv:\) \\
5 & \hspace{2cm} \text{True}() \to (P, \rho) \\
6 & \hspace{2cm} \text{False}() \to (Q, \rho) \} | tv \in testval-s \} \text{ in} \\
7 & \text{if } \text{card } res-s = 0 \\
8 & \text{then } \{(\text{mk-Divergent}(), \rho)\} \\
9 & \text{else } res-s \\
\end{array}
\]

\[
\forall (B, \rho_B) \in INF(mk-AgentIf(be, P, Q), \rho).
\]

\[
\begin{array}{c|c}
H & 10 \quad (B, \rho_B) \xrightarrow{\alpha} (B', \rho'_B) \\
C & 11 \quad (\text{mk-AgentIf}(be, P, Q), \rho) \xrightarrow{\alpha} (B', \rho'_B)
\end{array}
\]

Again we find a semantic function on \( \text{sts} \) elements in the first part of the diagram. On line 3 the set of values resulting from the evaluation of the \( be \) component is created which have a loose result, and can either be a truth-value or some other value. For each truth-value in the set \( testval-s \) a \( \text{sts} \) is added to the result set \( res-s \) (line 4–6). If the result set \( res-s \) is empty, which can only be the case if the values in \( testval-s \) are not boolean values, the resulting behaviour expression in the singleton \( \text{sts} \) set will denote divergence.

\[\Box\]
4.4.3.3 Standard action rules

These are the action rules for standard prefix, choice, composition relabelling and restriction for those cases that no looseness is involved and no time manipulation.

Example 4.3 (standard actions) A basic example of such a rule is the rule for prefixes with synchronization actions.

<table>
<thead>
<tr>
<th>rule Prefix-syn-action</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C 1 ( (\text{mk-Prefix}(\text{mk-SynAct}(lab, -), P), \rho) \xrightarrow{\text{lab}} (P, \rho) )</td>
</tr>
</tbody>
</table>

The diagram defines a single rule similar to the prefix rule for the standard SOS semantics for pure CCS.

4.4.3.4 Time manipulation

The rules that describe time manipulation fall in three groups: rules for the timed prefixes, rules for the idle prefix and rules for the idling choice and composition. The properties of the time manipulation are discussed in section 4.5.

Example 4.4 (time manipulation) The progression of time is related to actions labelled with a time value. E.g. the time progression during an idling prefix is described as follows.

<table>
<thead>
<tr>
<th>rule Prefix-syn-idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 ( \text{TDL}\varepsilon(tid, \rho) \Delta )</td>
</tr>
<tr>
<td>2 \ let ( t = \text{if } tid \in \text{dom } \rho.Cvenv )</td>
</tr>
<tr>
<td>3 \ then ( \rho.Cvenv(tid) )</td>
</tr>
<tr>
<td>4 \ else ( 0 ) in</td>
</tr>
<tr>
<td>5 ( \mu(\rho, Cvenv \leftarrow \rho.Cvenv \cup {tid \mapsto t + d}) )</td>
</tr>
</tbody>
</table>

| C 6 \( (\text{mk-Prefix}(\text{mk-SynAct}(lab, tid), P), \rho) \xrightarrow{\epsilon(d)} \) |
| 7 \( (\text{mk-Prefix}(\text{mk-SynAct}(lab, tid), P), \text{TDL}\varepsilon(tid, \rho)) \) |

The effect of taking an \(\epsilon(d)\) action is recorded in the current environment \(\rho\). \(tid\) is the time variable in a prefix with synchronization action. E.g. in \(a, \ast t \odot P\) the \(tid\) component is equal to the variable \(t\).
4.4.4 State manipulation

The rule template for state manipulating prefixes is a typical example of an environment manipulation template.

\[
\begin{array}{|c|}
\hline
1 & S E(mk-\text{Prefix}(mk-\text{SM}(\text{action}, d), P), \rho) \triangleq \\
2 & \text{let } s var = \text{dom } \rho . S \text{ env in} \\
3 & \text{let } env = (s var \triangleleft \rho . C \text{ env}) \cup \rho . S \text{ env in} \\
4 & \text{let } nenv-s = \{ nenv \mid (nenv, f, -) \in VS[\text{action}] \land f = \text{cont} \} \text{ in} \\
5 & \{(P, \mu(\rho, C \text{ env } \mapsto nenv, \\
6 & \quad \text{Stenv } \mapsto s var \triangleleft nenv)) \mid nenv \in nenv-s\} \\
\hline
\end{array}
\]

\[\forall \forall (B, \rho') \in S E(mk-\text{Prefix}(mk-\text{SM}(\text{action}), P), \rho) \]

There is no hypothesis. A state manipulation can not perform idle actions. The pattern \(d\) denotes a duration specification of the state manipulation itself. First the current state environment and value environment are selected. A set of environments is created (line 4), that resulted from the evaluation of \(\text{action}\). The statements in \(\text{action}\) are restricted to those that do not propagate exits out of their textual environment. Values that result from the evaluation of the statement, like e.g. call statements and return statements, are discarded. The new value-environment (line 5), and the new state environment (line 6) replace the current ones. The computation involved with the \(\text{action}\) component takes \(d\) time, which is realized by embedding the continuation behaviour \(B\) in a count-down timer, a component that consumes \(d\) time units and then behaves as its behaviour expression argument (as explained in section 4.5.1). Thus the followed scheme of actions can be summarized as follows:

1. copy \(Stenv\) to \(Cenv\)
2. manipulate \(Cenv\) resulting in \(Cenv'\)
3. extract the state from \(Cenv'\) and copy it back to \(Stenv\) giving \(Stenv'\)

In fact, maintaining a separate copy of the state up to date is only necessary in unit environments. In agent environments there is no interference as the state is manipulated in a single thread of control. In units the state is shared under all execution states, that may be composed to execute in parallel. Consider the next example that illustrates this point.
Example 4.5 In specification 4.6 unit \( X \) contains a state and two additional execution states, \( Y \) and \( Z \). Both operands in the composition \( Y \mid Z \) perform actions that updates the state component \( val \). As the whole statement \( \text{val} := \text{val} + y \) is encapsulated by an action construction, it is atomic in the sense that it cannot be pre-empted. Both execution states execute in a private environment \( \rho \) that share the same value for the state component \( \text{val} \). Whenever \( Y \) computes a new state value, the result must be recorded in the environment of \( Z \) and vice versa.

The semantics of the composition operator reflects these demands. Whenever the left operand of a composition takes an action, the following rule applies.

<table>
<thead>
<tr>
<th>rule Composition-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H )</td>
</tr>
<tr>
<td>( C )</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Whenever the left operand of the binary composition operator has an \( x \) transition on the environment of the composition (line 1), the whole composition has the same \( x \) transition (line 2,3). The possible side-effects due to a state-manipulation action are registered in the original environment \( \rho_q \) of the right operand. There is a similar rule for the right operand.

4.4.5 The interface function \( \nu S \)

The interface function \( \nu S \) connects the MOSCA structured operational semantics to the dynamic semantics of VDM-SL. It is defined through case
4.4. MOSCA’s transition system

analysis over the syntactic argument, which consists of a VDM-SL core abstract syntax domain value. The result is either a set of environments or functions that map environments to new environments. The definition of the interface function VS is given in specification 4.7.

\[
\begin{align*}
\text{VS : Definitions} & \rightarrow \text{ VENV-set} \\
\text{VS[defs]} & \triangleq \text{SemSpec(defs)} \\
\text{VS : Expr} & \rightarrow \text{ LEEval} \\
\text{VS[expr]} & \triangleq \text{EvalExpr(expr)} \\
\text{VS : Pattern} & \rightarrow \text{ LPatEval} \\
\text{VS[pattern]} & \triangleq \text{EvalPattern(pattern)} \\
\text{VS : LocalDef} & \rightarrow \text{ Ldef} \\
\text{VS[locals]} & \triangleq \text{let mk-LocalDef(vals, exprs, implfs) = locals in EvalLocalDef(vals, exprs, implfs)} \\
\text{VS : Bind} & \rightarrow \text{ BindEvals} \\
\text{VS/bind} & \triangleq \text{EvalBind(bind)} \\
\text{VS : Statement} & \rightarrow \text{ LSEval} \\
\text{VS/stat} & \triangleq \text{EvalStmt(stat)}
\end{align*}
\]

4.4.6 Environment manipulation

The transition rules are normally chained together to form inference diagrams (see e.g. [157]). These chains set out the justification for a transition of any behaviour expression. Within these diagrams the bottom inference justifies the transition step under inspection. Each level up forms the justification of the transition step below it. In figure 4.3 a general form of an inference diagram is shown for a straight chain of inferences. In the case of a sts in a conclusion with a synchronizing composition the chain splits, resulting in an inference tree. Each hypothesis one step up inherits elements of the environment of the conclusions through the succession of SE function applications. As such environment information is passed up the chain
Figure 4.3: Environment manipulation within inference chains

to determine the next justification inference. These inferences take place e.g. during construction of a derivation graph. An example of environment manipulation through the $SE$ functions can be found in example 4.15 in section 4.6.2.

Recall the structure of the environment such as defined in definition 4.3. The global value environment and the global agent environment are fixed once and forall during the evaluation of the definitions of a specification (see rule C.1.1 of appendix C). Initial $Venv$ is equal to $Genv$ and $Aenv$ is equal to $Gaenv$. The behaviour expression of a specification executes in $Venv$, which can be enriched by execution of prefix expression with in actions, let expressions and time variables from prefixes. The extend of the bindings depend on various factors, expressed by the next three rules.

1. The realization of a binding that involves a name already bound in the global value environment $Genv$, destroys the former binding and remains in effect until the global value environment is re-established.

2. The realization of a binding that involves a name already bound in
the actual value environment $Venv$, destroys the former binding and remains in effect until it is rebound itself, or until the a new actual value environment is constructed, which is done at each agent or unit service expression.

3. The bindings resulting from an agent-let expression are simply added to the current actual value environment. The occurrence of an agent constant within the body of an agent-let expression is simply replaced by the body of the constant without a change in the current environment.

The following example provides some cases to illustrate these rules.

$$A \langle v \rangle \triangleq a, \star t0 \odot$$

let $vn = f(v, t0)$ in

let $X = y, \star t1 \odot A \langle f(vn, (t0 + t1)) \rangle$

$$Y = \overline{Y} \odot \text{null in } X \mid Y$$

$$A' \triangleq \overline{a} \odot A'$$

The expression under investigation is the composition of $A$ and $A'$. The behaviour of the composition starts with a shared initial environment, schematically visualised as

$$\frac{f \mapsto m1,}{A \mapsto \ldots}$$

$$\frac{A' \mapsto \ldots}{\overline{A} \mapsto \ldots}$$

The behaviour of $A \langle 0 \rangle \mid A'$ evolves to:

$$(1) \left( A \langle 0 \rangle \mid A', \left. \begin{array}{c} f \mapsto m1, \\ A \mapsto \ldots \\ A' \mapsto \ldots \end{array} \right) \overrightarrow{AS_{ve}}$$

$$(2) \left( a, \star t0 \ldots \mid A', \left. \begin{array}{c} f \mapsto m1, \\ v \mapsto VS[0] \\ A \mapsto \ldots \\ A \mapsto \ldots \\ A' \mapsto \ldots \end{array} \right) \overrightarrow{AS_{uv}}$$

$$(3) \left( a, \star t0 \ldots \mid \overline{a} \odot A', \left. \begin{array}{c} f \mapsto m1, \\ v \mapsto VS[0] \\ A \mapsto \ldots \\ A \mapsto \ldots \\ A' \mapsto \ldots \end{array} \right) \overrightarrow{\rightarrow}$$
(4) \[
\begin{array}{c}
\frac{f \mapsto m_1, \quad v \mapsto V \mathbf{S}[0]}{A \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots} \quad \mapsto_{AVL}
\end{array}
\]

(5) \[
\begin{array}{c}
\frac{f \mapsto m_1, \quad v \mapsto V \mathbf{S}[0]}{t_0 \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots} \quad \Rightarrow_{AA}
\end{array}
\]

(6) \[
\begin{array}{c}
\frac{f \mapsto m_1, \quad v \mapsto V \mathbf{S}[0]}{A \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots} \quad \Rightarrow_{AS-\eta v}
\end{array}
\]

(7) \[
\begin{array}{c}
\frac{f \mapsto m_1, \quad v \mapsto V \mathbf{S}[0]}{t_0 \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots} \quad \mapsto_\iota
\end{array}
\]

(8) \[
\begin{array}{c}
\frac{f \mapsto m_1, \quad v \mapsto V \mathbf{S}[0]}{t_0 \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad v \mapsto \ldots}{A \mapsto \ldots} \quad | \quad \\
\frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots} \quad | \quad \frac{f \mapsto m_1, \quad A \mapsto \ldots}{A' \mapsto \ldots}
\end{array}
\]
4.5 Semantics of Time

\[
(9) \xrightarrow{\text{AS-ve}} \begin{cases} 
  f \mapsto m1, \\
  v \mapsto \forall S[f(\ldots)](\rho\ldots) \\
  A \mapsto \ldots \\
  A' \mapsto \ldots
\end{cases}
\]

The internal action \( \longmapsto \) results from synchronizations in compositions. Step (9) involves not only agent service but also the deletion of the null agent component from the composition.

4.5 Semantics of Time

The semantic rules in appendix C incorporate the manipulation of the time variables and define the meaning of the idle action.

4.5.1 Basic Properties

The basic properties of the semantics of time are summarized below.

1. **Maximal Progress** Whenever an agent can proceed by taking an internal action it will not wait.

   \[
   P \longmapsto P' \Rightarrow \exists d \in T \cdot P \xrightarrow{\text{c(d)}} P''
   \]

2. **Time Determinacy** To ensure the progress of time each agent, even the null agent and divergent agent can take idle actions. When time goes, if an agent idles, it can never reach different states:

   \[
   P \xrightarrow{\text{c(d)}} P' \land P \xrightarrow{\text{c(d)}} P'' \Rightarrow P \equiv P''
   \]

where \( \equiv \) means syntactical equality. Time determinacy results from the properties of idling in combination with and choice and composition:

\[
\begin{align*}
\text{idle}(t) \odot (P \oplus Q) & \sim \text{idle}(t) \odot P \oplus \text{idle}(t) \odot Q \\
\text{idle}(t) \odot (P \mid Q) & \sim \text{idle}(t) \odot P \mid \text{idle}(t) \odot Q
\end{align*}
\]

where \( \sim \) denotes an appropriate form of equivalence. The divergent agent \( \bot \) must be able to take idle actions as well. As time progression in compositions is synchronous, \( P \mid \bot \) would not allow the time to proceed if \( \bot \) would only take internal actions and no idle actions as well.
3. **Time Continuity** During idling no time moment may be passed without notice.

\[ P \xrightarrow{\ell(d+e)} P'' \Leftrightarrow P \xrightarrow{\ell(d)} P' \land P' \xrightarrow{\ell(e)} P'' \]

4. **Time persistency** By idling an agent will not loose the ability to perform an action that it is able to perform originally.

\[(P \xrightarrow{\ell(d)} P' \land P \xrightarrow{\tau} P'') \Rightarrow P' \xrightarrow{\tau} P''\]

These properties originate from the work on TCCS in e.g. [56], [158] and [220].

### 4.5.2 Semantic rules

The basic properties of time in **mosca** result from the incorporation of idling in the semantic rules of prefix, choice, composition, the null agent and divergent agents.

The null agent is inactive but can be engaged in idling, to enable time progression in compositions that involve a null agent. The divergent agent is either idling or busy with internal actions. A divergent agent must be able to idle, just like the null agent, to enable time progression.

The semantic rules for prefix constructions form the basis for time handling in **mosca**. There are three groups of rules.

1. **Prefix with synchronization, in or out action** This group consists of two rules for each specific action kind. The first rule states the effect of the specific action. The second rule expresses the effect of waiting through internal idling on the environment to synchronize.

2. **Prefix with state manipulation action** State manipulation is associated with an internal action. As such, to enforce the maximal progress property, state manipulation actions may not wait. There is no transition rule to enable a state manipulation action to wait.

3. **Prefix with idle action** This group consists of four rules. The first three describe the effect of an idling action. The first rule initiates the idling, the second rule describes idling in progress and the third rule describes the finalization of the idling action. E.g. the agent \( \text{idle}(5) \odot P \) operating in an environment \( \rho \) is after an initial internal action transformed into a construction called a count-down timer that contains the semantic evaluation of the idle expression, in this case 5, and the continuation behaviour \( P \). The count-down timer is only
4.6. Semantics of looseness

capable of taking idle actions. When the timer reaches zero, it transforms into a null-counter construction, which initiates the behaviour of \( P \).
The last rule describes the cumulative effect of a prefix with idle action when the continuation behaviour itself may idle also. E.g. the agent \( \text{idle}(5) \odot a, *t \odot Q \) may idle 5 time units and become \( a, *t \odot Q \) or idle more than 5 time units while waiting on the environment to synchronize on action \( a \).

The choice rules reflect the time determinacy property. In a choice expression such that both operands may idle, like in \( \text{idle}(5) \odot P + \text{idle}(7) \odot Q \) the choice as a whole may idle as much as the minimum of the two idle quantities. The idling of a choice expression is the source of the environment choice. Assume the following two initial idle transitions:

\[
(a, *t \odot P, \rho) \xrightarrow{\epsilon(d)} (a, *t \odot P, \rho \cup \{t \mapsto d\})
\]

and

\[
(b, *u \odot Q, \rho) \xrightarrow{\epsilon(d)} (b, *u \odot Q, \rho \cup \{u \mapsto d\})
\]

The choice

\[
((a, *t \odot P) + (b, *u \odot Q)), \rho
\]

is able to idle \( \epsilon(d) \) but the bindings of the two different time variables cannot be recorded into a shared environment. If after some idling one of the two operands is selected only one of the time variable bindings remains valid. Separate registration enables an easy selection of the valid binding.

4.6 Semantics of looseness

4.6.1 Loose value specification

The semantics of loose-specified constructions in VDM-SL is fixed to be either under-specified, or non-deterministic (see e.g. [15], where ARENTOFT and LARSEN give a first approximation of a dynamic semantics of VDM, [161], where MONAHAN highlights some consequences of under-specified specifications vs. non-determinism). The final exact meaning of loose-specified constructions in the dynamic semantics edited by LARSEN ([108]) is built around the following notions.

Definition 4.5 (Looseness in VDM-SL) Looseness is interpreted as erratic, which means that all possible computations, i.e. both terminating and non-terminating are taken into account. Loose VDM-SL functions are
interpreted as under specified, i.e. underdeterminatned, and operations are interpreted as nondeterministic.

First I will look into the semantic model for functions, followed by the semantic model of operations.

**Example 4.6 (looseness by a loose expression constructor)**

\[ Lfn1 : \mathbb{N} \rightarrow \mathbb{N} \]

\[ Lfn1(x) \triangleq \text{let } r \in \{1, 2, 3\} \text{ in } x + r \]

The function \( Lfn1 \) specifies a function that returns its argument augmented with either 1, 2, or 3. The looseness is the result of an inherent loose construction of VDM-SL, the let be construction.

The semantic denotations, that is, models for \( Lfn1 \) are simply given by the next set of denotations.

\[
\{\lambda env \cdot \lambda x \cdot x + 1 \\
\lambda env \cdot \lambda x \cdot x + 2 \\
\lambda env \cdot \lambda x \cdot x + 3\}
\]

Each element in this set is a loose value evaluator. Given a value environment \( \epsilon \) it computes a function that maps its single argument to itself augmented with either 1, 2 or 3. The set forms a loose expression evaluator. The denotations are bound to the function name \( Lfn1 \) in three environments each with a different denotation for function \( Lfn1 \), such as:

\[
VENV1 = \{Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 1\} \\
VENV2 = \{Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 2\} \\
VENV3 = \{Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 3\}
\]

In each environment the function \( Lfn1 \) behaves fully deterministic, i.e. the property \( Lfn1(\text{arg}) = Lfn1(\text{arg}) \) will hold for any argument of domain \( \mathbb{N} \).

**Example 4.7 (propagated looseness)**

\[ Lfn2 : \mathbb{N} \rightarrow (\mathbb{N} \times \text{Nat}) \]

\[ Lfn2(x) \triangleq (x, Lf1(x)) \]
4.6. Semantics of looseness

The function $Lfn2$ specifies a function that transforms its single number argument into a tuple constructed out of its argument, and its argument augmented with either 1, 2, or 3. Here the looseness results from propagation, i.e. from the application of a loose-specified function, $Lfn1$.

The semantic models for $Lfn2$ are similar to the models for $Lfn1$.

\[
\begin{align*}
VENV1 & = \{ Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 1, \\
& \quad \quad \quad Lfn2 \mapsto \lambda env \cdot \lambda x \cdot (x, Lfn1(cx)) \} \\
VENV2 & = \{ Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 2, \\
& \quad \quad \quad Lfn2 \mapsto \lambda env \cdot \lambda x \cdot (x, Lfn1(cx)) \} \\
VENV3 & = \{ Lfn1 \mapsto \lambda env \cdot \lambda x \cdot x + 3, \\
& \quad \quad \quad Lfn2 \mapsto \lambda env \cdot \lambda x \cdot (x, Lfn1(cx)) \}
\end{align*}
\]

Again, each environment binds the function names to deterministic functions.

Under-specification can possibly be used as deliberate omission of details which will be more efficiently stated later on in the development process.

**Example 4.8** (looseness through underspecification)

\[
Lfn3 \ (x : \mathbb{N}, \ y : \mathbb{N}) \ z : \mathbb{N}
\]
\[
\text{post} \ (x = 0 \land y = 0) \Rightarrow (z = 0)
\]

The function $Lfn3$ specifies a function which is zero at the origin. Nothing is said about possible other function values, which may not yet be of any importance at this stage in the development.

The semantic model for $Lfn3$ are environments, each containing a function from an environment to a function of two arguments, which has the property

\[
Lfn3(env)(0,0) = 0.
\]

The set contains countable many denotations for all such possible functions.

**Example 4.9** (extreme looseness)

\[
Lfn4 \ (x : \mathbb{N}) \ y : \mathbb{N}
\]
\[
\text{pre} \ \text{true}
\]
\[
\text{post} \ \text{true}
\]
Function \( Lfn4 \) models a total function that returns any value from the natural numbers. Here again the set of all possible denotations contains a countable number of models, each model consisting of one deterministic function.

\[ \text{Example 4.10 (erratic looseness)} \]

\[
\begin{align*}
Lfn5 \ (n: \mathbb{N}) & \ r: \mathbb{N} \\
\text{pre} & \ \text{is-Even}(n) \\
\text{post} & \ \text{true}
\end{align*}
\]

Function \( Lfn5 \) exhibits the most extreme looseness. The only meaning that can be given to \( Lfn5 \) is that, whenever it returns anything, it will have type \( \mathbb{N} \), and behaves just like \( Lfn4 \). If the argument does not fulfil the precondition, then it may return whatever natural number, or, erratically, the semantic value \( \bot \), which signifies infinite computation.

Operations can syntactically be distinguished from functions by two different phenomena. When specified explicitly, their bodies differ. Whereas functions have \textit{expression} bodies, operations have \textit{statement} bodies. When specified implicitly, the only difference is the specification of side-effects on statevalues through state manipulation.

Operations in VDM-SL are considered to be non-deterministic. According to Monahan ([161]), non-deterministic operations in abstract specifications allow \textit{latitude} in the implementation to extend the underlying concrete state upon which the implemented output could depend. To enable this latitude, the denotations of operations should include all possible refinements. This view leads to the current semantical model for operations: a \textit{relation}. Operations can be specified explicitly through a VDM-SL statement, or implicitly, through pre- and postcondition predicates. Each operation denotation is a \textit{set of quintuples}, each element in the set recording a relation between input values, state values before and after the effect of the operation, output values and a tag value marking whether the operation ended execution normally or by an exit construction. All values that involve the function space operator are excluded as input-, output- and statevalues.

\[ \text{Example 4.11 (looseness by loose expression constructors)} \]

Suppose we have the following operation definition:

\[
\begin{align*}
\text{LOPER1} \ (b: \text{B}) & \ r: \text{B} \times \mathbb{N} \\
\text{ext} & \ \text{rd} \ s : \mathbb{N} \\
\text{post} & \ \text{let} \ z \in \{1, 2\} \ \text{in} \\
& \ r = (b, z + s)
\end{align*}
\]
4.6. Semantics of looseness

Let $s$ contain the natural number 5. The denotation for operation $LOPER1$ is one relation consisting of the following tuples:

$\{(true, 5, 5, (true, 6), \text{cont}),$

$(true, 5, 5, (true, 7), \text{cont}),$

$(false, 5, 5, (true, 6), \text{cont}),$

$(false, 5, 5, (true, 7), \text{cont})\}$

The let construction results within the context op operations in non-determinism. The semantic addition to the environment is one binding of the identifier $LOPER1$ to the set containing the relation. □

4.6.2 Loose Agent specification

The basic behaviour operations like prefix, composition, restriction and choice enable the specification of loose behaviour, which is interpreted as nondeterministic.

Example 4.12 (nondeterministic choice) Consider e.g. the behaviour expression

$$(a \circ P) \oplus (a \circ Q)$$

which consists of a combination of prefix and choice. It is not obvious which operand of the choice should be selected on a synchronization on the action $a$. Either the transition

$$(a \circ P) \oplus (a \circ Q) \xrightarrow{a} P$$

will take place, or

$$(a \circ P) \oplus (a \circ Q) \xrightarrow{a} Q$$

□

It is not possible to force a choice here. When the underlying semantic model is a labelled transition system, there is no bias towards a specific selection. Each selection is acceptable, even e.g. always the selection of the first possible action found by examining a behaviour expression from left to right. It is obvious that such a selection procedure is far from being fair. The study of fairness issues within the behaviour of agents fixed through labelled transition systems has led to the development of fair transition systems (e.g. by MANNA and PNUELL in [149]). A general treatment of fairness in different semantical settings can be found in FRANCEZ’s work [74].
Example 4.13 (nondeterministic composition) Another source of nondeterministic behaviour can be found in the composition operator. Consider the following process definitions (based on an example in [157]).

\[
X \\
\text{ports} \; \text{syn} \; x \\
X \triangleq x \circ Y
\]

\[
Y \\
\text{ports} \; \text{syn} \; y \\
Y \triangleq y \circ X
\]

The behaviour of agents \(X\) and \(Y\) are infinite and symmetrical. The agent \(X\) evolves to \(Y\) after synchronization on \(x\), and \(Y\) evolves to \(X\) after synchronization on the action \(y\). The agent \(Z\) may synchronize on \(b\) with another agent, or perform an internal action by synchronizing on the action \(a\). Let \(\rho\) be the environment component of the sts in which the system is on arriving at the behaviour expression bound to \(Z\), and it is assumed that all actions are environment preserving. External synchronization on action label \(b\) then leads to

\[
(((a \circ X) \mid ((b \circ \text{null}) \oplus a \circ Y)) \setminus \{a\}), \rho) \xrightarrow{b} (((a \circ X) \mid \text{null}) \setminus \{a\}), \rho)
\]

resulting in a state, from which no further transitions are possible. The only prefix is the expression, \(a \circ X\) is inhibited by the restrictor \(\setminus \{a\}\). Internal synchronization on the action \(a\) leads to

\[
(((a \circ X) \mid ((b \circ \text{null}) \oplus a \circ Y)) \setminus \{a\}), \rho) \xrightarrow{a} (((X \mid Y) \setminus \{a\}), \rho)
\]

from which both \(x\) and \(y\) transitions are possible. When we depict the possible transitions as derivation sequences, a tree (the derivation tree) evolves, with as nodes the sts configurations and labelled branches for each possible derivation. Often the evolving tree can be folded into a graph, by merging paths with shared nodes. The input and output action labels
may be taken from infinite sets, like the natural numbers \( \mathbb{N} \), which makes it impossible to draw complete derivation trees. Transitions due to these actions are represented by branches that are labelled with a generic action label rather than with the actual action label. The derivation graph for \( Z \) is given in figure 4.13.

\[
(((a \odot X) \mid (b \odot \text{null}) \oplus a \odot Y) \setminus \{a\}),\rho
\]

\[
\xrightarrow{l}
\]

\[
(((X \mid Y) \setminus \{a\}),\rho)
\]

\[
\xrightarrow{b}
\]

\[
(((a \odot X) \mid \text{null} \setminus \{a\}),\rho)
\]

\[
\xrightarrow{x}
\]

\[
(((Y \mid Y) \setminus \{a\}),\rho)
\]

\[
\xrightarrow{y}
\]

\[
(((X \mid X) \setminus \{a\}),\rho)
\]

\[
\xrightarrow{x}
\]

\[
(((Y \mid X) \setminus \{a\}),\rho)
\]

\[
\xrightarrow{x}
\]

\[
(((X \mid X) \setminus \{a\}),\rho).
\]

\[
(((Y \mid Y) \setminus \{a\}),\rho)
\]

\[
(((X \mid X) \setminus \{a\}),\rho).
\]

Figure 4.4: Derivation graph for agent \( Z \)

Examining the derivation graph, two nondeterministic sts configurations appear,

\[
(((Y \mid Y) \setminus \{a\}),\rho)
\]

and

\[
(((X \mid X) \setminus \{a\}),\rho).
\]

From the first sts two synchronizations on \( y \) are possible, resulting in different sts configurations. From the second sts, two different \( x \) synchronizations are possible.

\[\square\]

Example 4.14 (propagation of looseness through composition) Suppose we have two other agents, \( P \), with associated behaviour \( x \odot Q \) and \( Q \) with associated behaviour \( y \odot P \). Now the system \( (P \mid Z) \) enters after two
deterministic internal actions the sts

\[((Q \mid (Y \mid Y) \setminus \{a\}), \rho)\]

which has two internal action transitions,

\[((Q \mid (Y \mid Y) \setminus \{a\}), \rho) \xrightarrow{t} ((P \mid (Y \mid X) \setminus \{a\}), \rho)\]

and

\[((Q \mid (Y \mid Y) \setminus \{a\}), \rho) \xrightarrow{r} ((P \mid (X \mid Y) \setminus \{a\}), \rho)\]

Again there is no bias to prefer one possibility over the other. \(\square\)

Another cause of nondeterministic behaviour results from loose value specification.

Example 4.15 (loose value nondeterminism) Suppose we have the following schematic agent definition

\begin{align*}
OneOrTheOther (B) \\
\text{ports} & \text{ syn a} \\
\text{syn b}
\end{align*}

\begin{align*}
OneOrTheOther (true) & \triangleq a \circ Bexpr_1 \\
OneOrTheOther (false) & \triangleq b \circ Bexpr_2
\end{align*}

The behaviour of the agent is determined by the value of the value part. Whenever true it behaves like \(a \circ Bexpr_1\), otherwise it behaves like \(b \circ Bexpr_2\). The agent behaves deterministic. Suppose the agent is subject to an agent service in the context of specification 4.9. Then the resulting

\[
\begin{array}{|c|c|}
\hline
\text{Choice} & \triangleq \text{let } v \in B \text{ in } \\
OneOrTheOther (v) & 4.9 \text{ Value-let-be looseness} \\
\hline
\end{array}
\]

behaviour expression is not deterministic. It's behaviour depends on the actual value of \(v\), which can be either true or false. The evaluation of the let expression results in two environments, one containing the binding \(v \mapsto \text{true}\) and the other containing \(v \mapsto \text{false}\). Again assuming an environment \(\rho\), the single sts

\[((\text{let } v \in B \text{ in } OneOrTheOther (v)), \rho)\]

is transformed by value-evaluation into a set of sts's

\[
\{(OneOrTheOther (v), \rho \cup \{v \mapsto \text{true}\}), \quad (OneOrTheOther (v), \rho \cup \{v \mapsto \text{false}\})\}
\]
4.6. Semantics of looseness

each element in the set containing one specific enrichment of the original environment \( \rho \). After the evaluation of the agent service expression through pattern matching, the following set of sts’s results:

\[
\{(a \circ Bexpr_1, \rho), \\
(b \circ Bexpr_2, \rho)\}
\]

in which each sts is a valid starting point for further transitions. The derivation graph becomes

\[
\begin{array}{c}
((\text{let } v \in B \text{ in } OneOrTheOther \langle v \rangle ), \rho) \\
\end{array}
\]

The behaviour of the agent value let-be expression can be viewed as the merged behaviours of all possible value let expressions with deterministic value binding. The looseness here is of another nature than the looseness resulting from the nondeterministic behaviour operators. Here fairness arguments will not influence the selected transitions, as in the latter case. Which transitions will be selected in an implementation depends on reification of the value components in the behaviour expression. When, in the example, after reification the let-be expression is transformed e.g. into a deterministic let construction, the choice of transition-path is narrowed accordingly.

Example 4.16 (loose pattern matching on value parts) Whenever an agent is defined through a series of behaviour definitions, the possibility of loose behaviour occurs. In specification 4.10 the agent UpOrDown either increments its value part or decrements it. The patterns to select one of these two, are equivalent. After all, both \( p \) and \( q \) match the whole set of natural numbers. The sts \((UpOrDown \langle n \rangle, \rho)\) is first transformed into a set of sts’s

\[
\{(\overline{\text{next}}(p) \circ UpOrDown \langle p + 1 \rangle, \rho \cup \{p \mapsto n\}), \\
(\overline{\text{next}}(q) \circ UpOrDown \langle q - 1 \rangle, \rho \cup \{q \mapsto n\})\}
\]

which forms the basis for two different transitions on the same action \( \overline{\text{next}} \).
\[
\begin{align*}
UpOrDown (N) \quad & \text{4.10 non deterministic patterns} \\
\text{ports out } & \text{next} \\
UpOrDown (p) & \triangleq \text{next}(p) \odot UpOrDown (p + 1) \\
UpOrDown (q) & \triangleq \text{next}(q) \odot UpOrDown (q - 1)
\end{align*}
\]

**Example 4.17 (shared device manager)** This example highlights once more the effect of looseness on the behaviour of systems that are build from compositions. Let Semaphore be the agent defined in 3.2 and MCell be defined as in 3.4. Suppose we have two user processes that both acquire the service of the memory cell. To exclude access to a memory cell that is busy in service the semaphore is acting as an arbiter. The User agent repeatedly reads a value and writes it updated back. It is parameterised on the updating function. Let \( f : N \rightarrow N \) and \( g : N \rightarrow N \) be two functions defined in the environment of the agent System. The whole system is defined in spec. 4.12 It’s pictorial form is given in figure 4.5. The sts formed from a

\[
\begin{align*}
\text{User} (N \rightarrow N) \quad & \text{4.11 User} \\
\text{ports in } & \text{read : } Z \\
\text{out } & \text{write : } Z \\
User (f) & \triangleq p \odot \text{read}(x) \odot \text{write}(f(x)) \odot v \odot \text{User}
\end{align*}
\]

\[
\begin{align*}
\text{System} \triangleq \\
\text{let User1 = User \( (f) \)} \text{ in} \\
\text{let User2 = User \( (g) \)} \text{ in} \\
\quad (User1 | User2 | Semaphore | MCell (0) ) \\
& \quad [\text{read} \leftarrow \text{get}, \text{write} \leftarrow \text{set}]
\end{align*}
\]

given environment \( \rho \) and the behaviour expression defined by the System agent,

\[
((User1 | User2 | Semaphore | MCell (0) )[\text{read} \leftarrow \text{get}, \text{write} \leftarrow \text{set}], \rho)
\]
is a starting point for two \( i \) transitions, both resulting from a synchronization on the \( p \) action. The first one involves User1 and the second one involves User2. The system exhibits a circular behaviour (fig. 4.6). Whenever the system returns to the starting state System either one of the
two alternatives must be selected. Again here fairness constraints will enforce a selection scheme where both alternatives are selected just as many times. This fairness, however, is not present in the semantics of MOSCA and should be incorporated later on during process refinement. The actual behaviour expression components of the sts states are:

\[
\begin{align*}
System &= User_1 \ | \ User_2 \ | \ Semaphore \ | \ MCell \ (p) \\
User_1-A &= (\text{read}(x) \odot \text{write}(f(z)) \\
&\quad \odot v \odot User_1) \ | \ User_2 \ | (v \odot Semaphore) \ | \ MCell \ (p) \\
User_1-R &= (\text{write}(f(z)) \odot \\
&\quad v \odot User_1) \ | \ User_2 \ | (v \odot Semaphore) \ | \ MCell \ (p) \\
User_1-W &= (v \odot User_1) \ | \ User_2 \ | (v \odot Semaphore) \ | \ MCell \ (f(p)) \\
User_2-A &= User_1 \ | (\text{read}(x) \odot \\
&\quad \text{write}(g(x)) \odot v \odot User_2) \ | (v \odot Semaphore) \ | \ MCell \ (q) \\
User_2-R &= User_1 \ | (\text{write}(g(x)) \odot \\
&\quad v \odot User_2) \ | (v \odot Semaphore) \ | \ MCell \ (q) \\
User_2-W &= User_1 \ | (v \odot User_2) \ | (v \odot Semaphore) \ | \ MCell \ (g(q))
\end{align*}
\]
4.6.3 Loose Time Specification

The time consumption of idle actions is specified by time expressions, i.e. value expressions that after evaluation result in a real value. Within these expressions looseness may occur due to loose value specification. Let $T = \text{Real}_0^+$.  

Example 4.18 (time looseness by loose value specification) Let $P$ denote an agent defined in the environment $\rho$.

$$\text{IdleAndThen}P \triangleq$$

let $tc : T$ be s.t. $tc \leq 10.0$ in

$$\text{idle}(tc) \circ P$$

The agent $\text{IdleAndThen}P$ denotes a behaviour expression that will idle for $tc$ time units and then become $P$. The value-let-be construction delivers environments consisting of one binding between $tc$ and a time value

$$\{ tc \mapsto \text{bindval} \mid \text{bindval} : \mathbb{R} \cdot \text{bindval} \geq 0 \land \text{bindval} \leq 10 \}.$$  

Each environment initiates a time consumption equal to the value bound to $tc$. According to transition rule C.3 the conclusion containing the sts

$$(\text{IdleAndThen}P, \rho)$$
will appear in a set of transition rules quantified by

\[
\text{let } \text{beh}_\text{env} = \{ \text{let } tc : T \text{ be s.t. } tc \leq 10.0 \text{ in } \text{idle}(tc) \otimes P, \\
\quad \rho \cup \{ tc \mapsto \text{bindval} \} \mid \text{bindval} : R \cdot \text{bindval} \geq 0 \wedge \text{bindval} \leq 10 \} \text{ in} \\
\forall (Bexpr, \rho_1) \in \text{beh}_\text{env} \quad (Bexpr, \rho_1) \xrightarrow{\delta} \ldots \\
(\text{IdleAndThenP}, \rho) \xrightarrow{\delta} \ldots
\]

where $\delta$ may take any value less or equal to the value bound to $tc$. Thus, $\text{IdleAndThenP}$ specifies an agent that may idle for a fixed amount of time bound by the interval $[0, 10]$ and then behaves like $P$. \hfill \Box
5.1 Introduction

The question with respect to the hypothesis of this research effort cannot be resolved without first deciding on the particular qualities of both VDM-SL and CCS with respect to software development in general. These issues are covered here.

This chapter introduces a simple framework for software development that serves as reference model in the following chapters of the thesis. To picture MOSCA the context of the proposed framework answers are sought on the following two questions.

1. What kind of software systems can be described most effectually with the MOSCA notation?

2. Given the particular view on software development contained in the software development framework, where does MOSCA fit in?

The answers to these questions are not easy to find. One specific approach to answer the first question is discussed in section 5.2. The framework for software development is presented in section 5.3. The section opens with a subsection that sketches a simple phase model for the software life cycle. Next I present a simplified model for software development based on three different activities: specification, implementation and verification. Each activity is shortly introduced. Section 5.3.3 reviews different models that can be used within the software development activities and offers criteria that may help to form a judgement whether a modelling notation is adequate for its proposed task. These software development activities are used as stepping stones for the placement of VDM and CCS in the framework of software development. The general viewpoints of the rôle of VDM and
CCS in the software development activities are summarized in the closing section of this chapter, in which I postulate a rôle for MOSCA in the software development activities. In the closing chapter of the thesis I will review this topic with the help of the presented material in the following chapters.

5.2 On finding the intended class of systems

The answer to the first question enables the answer on the second question and more important, it provides the ground to answer topic 3 (a) of the validation suite.

Given the fact that MOSCA is already constructed and ready for trial, there are several ways to find answers on the first question ([226]).

- Try the language on a suitable set of example problems, and evaluate the resulting specifications on readability and conciseness criteria. From enough experience gained in this way, it should be possible to extract general principles that point to a specific class of systems. The obvious problem raised by this approach is to find a suitable set of example problems. ZAVE [226] advises to stick to well defined problems, that are documented in general publications. Inventing your own problems may limit your results severely. Tackling published problems opens the possibility of comparison to specifications resulting from other approaches. It is not wise to include realistic examples from industry projects, as the size of these cases often inhibits a complete treatment and in the case of a mismatch between the notation and the case study much time is wasted.

- Draw a set of correspondences between language features and characteristics of systems in specific classes. This way leads to a set of systems for which the notation has a natural affinity.

To find (or propose) a specific class of systems for MOSCA I have taken a joint approach. First I have made associations of system characteristics with MOSCA features and then made a selection of the resulted characteristics, and then looked for appropriate cases on which MOSCA could be tried. Figure 5.1 summerizes the correspondence analysis.

Both functional and state-based data manipulation originate from VDM-SL and have been used extensively to specify state-based and sequence based systems. The availability of cyclic processes and the basic CCS behavioural operators correspond with continuously running concurrent systems.
5.2. **On finding the intended class of systems**

![Diagram](image)

Figure 5.1: Correspondance between language features and systems

The corresponding arrow between timed actions and the delay action features of MOSCA and the class of systems with real-time data interfaces indicates the intended functionality of the time extensions of MOSCA.

The systems indicated in the last correspondence have a dynamical nature, with the restriction that the interface between the components have a fixed potential structure. All possible entities that may participate in the interface must be known beforehand. Predefined connections between entities in interfaces of different components may be dynamically created or destroyed.

I have taken representative examples of two specific classes: continuously state based concurrent systems and concurrent systems with real-time data interfaces. The first example features protocol specification and covers a small part of a system that realizes the inter node communication in computer networks. The second case presents a process control system in which timing plays an important rôle. The cases are presented in chapter 7.

Before the the second question, related to the specific aspects of the software components that can be specified, can be answered satisfactory the software development framework is shortly introduced.
5.3 Software development

There is neither in the scientific literature nor in the practice of computer systems engineering a strong consensus with respect to phases or activities in the development process of computer systems. At the present time there is not any single generally accepted technique or tool. There is, although, a set of paradigms that have matured well enough and are recognized as specific development methods. Each paradigm covers a particular range of activities in the software life cycle, the succession of activities that starts with the users problem and ends with the death of the software product.

5.3.1 A simple software life cycle model

Freeman and Wasserman (in their IEEE Tutorial on Software Design Techniques [75]) have captured the evolution of software design approaches from the early seventies up to the eighties. Cooling in [58] summarizes the software life cycle in a more recent perspective as follows. The life cycle may be divided into 6 stages, which are:

1. Statement of Requirements (SOR) In the first phase the customer's needs are stated and captured in the customer's requirements.

2. Software Requirements Analysis and Specification — Here the purpose is to establish precisely what the software components in the system are supposed to do. The activity completes with a complete specification of the software components to be built: the software requirements specification or SRS.

3. Architectural design — This stage is concerned with modelling the internal software structure, using information supplied in the software requirements specification. It defines the essential software components of the system, how these fit together and how they communicate.

4. Physical Design — (also often called detailed design). Here the architectural structure arrived at in the previous stage is partitioned to fit onto hardware. In multiprocessor designs and distributed systems this is a critical activity.

5. Coding — Ultimately the software tasks are expressed as a set of sequential single-thread program structures. The function of the coding stage is to take these design structures and translate them into source code.
6. *Test, integration and debug* — The purpose here is to show that the finished code performs as specified. It involves testing of individual software modules, combined modules and the whole system.

The phases are linked together to form the life cycle. The phases are in general subdivided into specific parts which can be iterated over. The whole life cycle contains also various feedback loops that enable iteration over various parts of the life cycle. Phases can be extended to hold explicit prototyping activities and may be concentrated to develop specific hard parts of the desired system instead of developing the whole system in one big effort. Examples of such life-cycle models are the prototyping models of BOHR [39], ANDRIOLE [13] and DAVIS, [63]. The evaluation of risk has been the basis for BOEHM's spiral model [41].

### 5.3.2 Software Development Activities

An elegant conceptual breakdown of all these life cycle models is the observation that the software development process can be regarded as sequences of development steps. This approach is advocated by e.g. VAN KATWIJ in [123]. In each step a transition is made that is directed to the final implementation of the software system. A development activity can be characterised by four different elements:

1. *the source specification* that describes the starting point of the development activity,

2. *the target specification* that forms the result of the development activity,

3. *the development activity* that transforms the source specification into the target specification and

4. *the verification activity* that consists of establishing the conviction that the target specification is correct with respect to the source specification.

For the sake of simplicity the whole range of development steps can be squeezed together into two major steps: specification and implementation. The specification step starts from the user requirements and results in the SRS that forms the basis for the design phases. The implementation step includes the different design phases and coding phase and results in an implementation.

Figure 5.3.2 captures the development activities schematically. The three activities together will be denoted by the acronym SDA, which stands for the Software Development Activities.
5.3.2.1 Specification

The Specification activity spans both the software requirements analysis (problem analysis) and software requirements specification (product description) phases. These two phases are often strongly interwoven, and often incremental in realization. The activities of the two phases are quite distinct, however. During problem analysis the understanding of the problem is sought, by e.g. delineating and refining of constraints, trade-offs between different conflicting constraints, expanding information of the problem etc. The main problems in this part of the specification activity are finding ways of trading-off constraints and organizing the expanded information on the problem. During product description the hard decisions are made that result in the conception of the product: conflicting views are resolved, inconsistencies and ambiguities are eliminated.

The source specification for this development step in the informal specification of the user’s needs. The activity is mostly occupied with establishing what the software product should do. The target specification is the SRS. The SRS is divided into two divisions:

- a division that states the behavioural requirements of the product, and

- a division that states the non-behavioural requirements of the product.

The behavioural requirements define what the system does. These describe
all the inputs and outputs to and from the system as well as information concerning how the inputs and outputs interrelate.

The non-behavioural requirements define the attributes of the system as it performs its job. They include e.g. a description of the systems levels of efficiency, reliability, security, maintainability, portability, etc (see e.g. the survey paper of Boehm [40]).

5.3.2.2 Implementation

The *Implementation activity* spans the design and coding phases. These phases can be viewed as one with respect to their ultimate target: the software product itself. The source specification for this activity is the SRS, the target specification is the actual code of the implementation. This activity is mostly occupied with establishing *how* the software will perform the tasks that are specified in the SRS.

5.3.2.3 Verification

The *verification activity* falls apart in two separate activities: verification in the specification and verification in the implementation activities.

The verification activity during specification is mainly concerned with consistency checking, i.e. verifying the SRS against being ambiguous, incomplete, redundant or finding inconsistencies with respect to the user requirements.

The verification activity during implementation is mainly concerned with the verification of the different descriptions with respect to the SRS and the verification of the non-behavioural requirements as stated in the SRS.

5.3.3 Models in the SDA

The various documents produced during the two development activities present *models* of the desired system on different levels of abstraction. These models serve a specific rôle in the system development.

5.3.3.1 Models in Specification

From the development's point of view the ultimate purpose of the system is described by the collection of user's requirements. These are often stated in informal text, collected in the statement of requirements phase. From the user's requirements the software requirements evolve during the software requirement analysis, which are the basis for deriving software requirements.
The modelling activity can be undertaken on various description levels. On the outmost level there is the entire software system in its context. The description of a model for a software system on this level should contain:

- the functionality of the whole system,
- the interfaces of the system with its environment,
- the conditions of the environment under which the system will operate.

The main characteristics of this model are the distinction between system and environment and the perfect internal technology. The model is often denoted with the term environmental model\(^1\). The model as such defines a border between entities outside of the system and the system itself. E.g. in the case of process control systems, the control system is inside the system under development, and the controlled system, including the devices that enable the control, forms part of the environment.

Desired features of the system under development that are in general not captured in this system model are the non-behavioural requirements. The non-behavioural requirements such as timing constraints and hardware constraints have a strong impact on the implementation activity and should be stated as clearly as possible. Decomposition of the environmental model delivers the behavioural model(s) of the software requirements.

Requirements on techniques for the description of behavioural models during the specification activity are nicely captured by DAVIS in [63]. He distinguishes 10 different traits, which are summarized below.

1. Proper use of the technique should reduce the ambiguity of the SRS.

2. Proper use of the technique should result in an SRS that is helpful to and understandable by customers and users who are not computer specialists.

3. Proper use of the technique should result in a SRS that can serve effectively as the basis for design and test.

4. The technique should provide automated checking for ambiguity, incompleteness and inconsistency.

5. The technique should encourage the requirements writer to think and write in terms of external product behaviour, not the products internal components.

\(^1\)This term originates in YOURDON's Structured Analysis.
6. The technique should help organize information in the SRS.

7. The technique should provide a basis for automated prototype generation and automated system test generation.

8. The technique should facilitate SRS modification.

9. The technique should permit annotation and traceability.

10. The technique should employ underlying models that facilitate the description of a system’s external behaviour within the intended application environment.

These traits are not easy to gather in one technique all at once. Such a technique would be the ultimate SRS specification notation, and the existence of such a technique is doubtful. However, the list of traits forms a basis for the judgement whether a given notation can be judged to be a SRS notation.

5.3.3.2 Models in Implementation

The inner structure of systems is composed of various parts. The initial decomposition of the whole system results in a model that reveals the inner structure of the whole system and should contain descriptions for:

- the structure of the whole system, in terms of physical parts and their interconnecting links,

- the rôle of each part, with respect to the functionality of the whole system,

- the rôle of each part, with respect to the behaviour of the whole system, including performance aspects, reliability and failure behaviour.

This model may be characterised as the system architecture model. The architecture is in general the starting point for the implementation of the system. By further decomposition and refinement of the various parts implementation models are created from which the implementation of the system gradually evolves.

Desirable properties of notations for architectural models and implementation models can be summarized by the next four general criteria:

1. Power of expression. These techniques should allow the expression of a wide range of interesting properties that are relevant to the description of the system of choice. E.g. for the design of distributed systems concepts such as communication and concurrency are strongly desired.
2. Well-definedness. These techniques should be based on solid mathematical foundations to enable the determination of the meaning of a model unambiguously. The foundation also should form the basis for theories for verification.

3. Abstraction facilities and modularity. These techniques should offer constructions for hiding irrelevant details and for decomposition of models into meaningful components.

4. These techniques should offer decomposition and refinement facilities under control of a verification facility, such that a decomposition or refinement can be checked with respect to a formal relationship between the starting point of the decomposition or refinement and the result.

5.3.3.3 Models in Verification

A description of a model should enable the verification of certain parts or system properties, either

- by static analysis of the description or

- by dynamic analysis through simulation or rapid prototyping of the specification.

Static analysis is often done by detailed inspection of the descriptions. If certain properties of the descriptions are formally stated, axiomatic methods may be used to inspect their validity. E.g. safety or liveness properties of a description expressed in a temporal or time logic can be proved to hold, a fact that is not achievable by any testing, irrespective of the length of the tests. To derive formally stated properties of a description, the description itself must be based on a notation with a formal semantics. The verification of the timing constraints of a system forms the central issue in the verification of real-time systems. Without a formal statement of these constraints, the main aspect of these systems remains uncheckable! The ability of dynamic analysis through simulations or rapid prototyping depends also strongly on the properties of the notation used to describe the models. Again a formal semantics is needed to give a model a precise meaning, which then is taken as basis for simulation or translation into a prototype.

Modelling and verification are related issues. The very nature of the modelling concept strongly influences the verification techniques that can be applied on the constructed model. What properties or parts of the system can be verified depends also on what properties or parts are modelled.
5.4 The models of VDM, CCS and TIME

The following three sections provide a short overview of some aspects of VDM-SL, CCS and time with respect to software development in general. Each subsection discusses the rôle of VDM, CCS and time in the three different software development activities. The section on VDM provides some personal viewpoints based on [180]. The section on time is partly based on [163].

5.4.1 VDM in software development

The Vienna Development Method is more than just a specification notation. It provides a scala of paradigms to develop specifications and to refine these specifications into code. Jones has developed the methodological aspects substantially by providing refinement rules that relate implicit and explicit forms of specification ([115]). Björner has provided several views on the methodological aspects of VDM ([36], [32]). Most of these observations provide guidelines for the development of parts of systems, but lack detailed information on the application of VDM within the whole range of software development activities. Often the rôle of VDM in the early phases of the life cycle model (phases 2 - 3) is left more or less obscure. The publications do seldom provide any specific details on the application of VDM during these early activities.

In [180] Plat c.s. have investigated the rôle of VDM in the software development activities that are covered by the software development model DoD-STD-2167A ([4]). This model covers all activities in the presented life cycle model. Our findings are summerized in the next three subsections.

5.4.1.1 VDM in Specification

The support of VDM in the development of the software requirements specification and the associated analysis of the requirements is rather limited. VDM offers no specific support to develop environmental models. During the requirements specification phase only little is known about the whole state of the system. The application of VDM here mostly results in separate specifications of isolated components of the system. Integration of these specifications leads often to dense specifications that are not easily to comprehend ([177]). VDM is not equipped to specify non-behavioural requirements ([55]).

There is currently much research effort directed at the integration of
informal and formal methods to provide at least rigorous methods for software requirement analysis and specification (SVDM LIT).

5.4.1.2 VDM in Implementation

During the design phases the true power of VDM emerges. The notation provides powerful abstraction primitives to specify data components on various abstraction levels, ranging from highly abstract mathematical data models to low-level implementation-oriented data types. Further the notation provides operational abstractions dedicated to specify components containing single control threads. The support on the architectural design phase is somewhat limited due to the absence of a strong structuring mechanism. The strong decomposition and refinement properties of the notation give ample ground for a very successful application of VDM-SL during the subsequent design activities.

5.4.1.3 VDM in Verification

VDM-SL has been given a formal logic proof system based on LPF, the logic of partial functions ([23]). This logic enables the static verification of various aspects of single specifications as well as relations between two specifications. In [71] a case study is presented that covers the verification during design of a specification in VDM-SL aided by a tool called mural.

The rôle of VDM-SL in dynamic verification is restricted to its executional subset. Although there are several experiments and product development activities going on (see e.g. [185]), it is yet too early to judge the quality of the rôle of VDM-SL in simulation or rapid prototyping.

5.4.2 CCS in software development

In this section CCS is placed within the three activities specification, implementation and verification.

5.4.2.1 CCS in Specification

CCS is a highly abstract notation that offers constructions to describe communication and concurrency. Its constituents are simple but extremely powerful: terse notation, equational characterization, expansion theorem and equivalence relations. The notation is perhaps a little too powerful to be commonly used in practical settings.

The CCS model is event-based. An event is experienced by an agent through synchronization. The interleaving semantics of CCS define a total temporal ordering of events in a CCS specification. The processes specified
by CCS agents have no internal state and their behaviour is specified in terms of event/action constructions (the prefix operator) non-deterministic choice, and composition. Different levels of abstraction are easily created through the hiding and renaming operators. These facilities makes CCS in particular applicable in the area of (protocol specification for) communication networks, distributed systems, or systems that are in some way physically partitioned and where the constituents can communicate.

The formal semantics makes the notation especially suitable for application areas where reliability is very important. The software requirements specifications for these areas should clearly be unambiguous and well understood. The understanding of a CCS specification can be enlarged by verification of specific properties through rigorous inspection or by simulating the behaviour of the specification. Simulation can be performed by interpretation of the SOS semantics and the application of the equational characterization. An example of a combined tool that supports the development of CCS specifications is the Edinburgh Concurrency Workbench [159]. The SOS semantics can also be used as starting point for the automatic generation of a rapid prototype.

5.4.2.2 CCS in Implementation

The SOS semantics provide a means to describe the behaviour of CCS agents in an operational setting. As observed above this enables the construction of simulators and rapid prototypes of the specifications. During the design phases both hardware and software specifications are transformed into actual hardware and software components.

The transformation of CCS specifications into software components faces the problem of logical process elimination. Logical processes denote the processes on the abstraction level of specification. Physical processes denote the processes on the implementation level. Physical processes have a private thread of control and execute in parallel with other physical processes in an implementation. They are the realizations of the logical processes in a specification. Remark that in CCS each process expression denotes a separate process. The relation between physical and logical processes can be characterised in two different ways: an one-to-many relation between physical and logical processes and an one-to-some relation.

The one-to-many relation is linked with the problem of elimination of processes. To implement all the logical processes as physical processes in an implementation will almost certainly withhold the implementation of meeting its performance requirements. Implementations of implementation languages that support the realization of more than one physical process add in general elaborate run-time support systems to the translated code.
to control the parallel execution of the physical processes to enable communication. The control may further depend on the services of the operating system under which the translated code executes. A direct transformation of a CCS specification into an implementation in this context is not feasible. The problem can be tackled in two different ways:

- the CCS specification is refined into more concrete CCS specifications that contain less logical processes until the number of logical processes more or less matches the number of desired physical processes; or

- the design starts from scratch with an initial model based on logical *modules* which is then refined and at some point in the design transformed into a set of physical processes by partitioning of the set of logical modules.

In the second solution the CCS specification can be used as an executionable oracle against which the correctness of the system can be checked.

In the first solution the refinement can be done under control of one of the equivalences up to the step where the CCS specification is transformed into implementation language code. This step coincides with the other case of physical-to-logical process relations. If there is an one-to-some relation between the physical and the logical processes such that the total number of logical processes matches more or less the total number of desired physical processes, the direct transformation of a CCS specification into an implementation is feasible. Here the implementor is faced with the problems of transforming the semantics of non-deterministic choice and composition into constructions that are supported in the implementation language.

The extend of this problem is strongly influenced by the nature of the constructions that support concurrency on the level of the implementation. For example, in the case of CSP a whole new implementation language and execution platform was developed to support the transformation of CSP into efficient executing code. This language is OCCAM and the execution platform is the Inmos Transputer. Apart from this language there are several other languages specially dedicated to the asymmetrical synchronous communication mechanism of CSP, such as CSP-S, CSP/80, ECSP, Pascal-m etc. All these languages are covered in the survey paper of BAL c.s. in [21].

In CCS the communication is, like in the CSP language, also synchronous but unlike CSP symmetrical. The two processes involved in a synchronization are not aware of each other's identity, whereas in CSP the server/client relation between the processes is applied. A client process must always be aware of the identity of the server. The symmetrical rela-
tionship poses extra problems on the transformation and may be the main cause that no languages are developed that support the CCS approach.

The conclusion given here with respect to the applicability of CCS in the implementation stages is based on the above observed facts. CCS is certainly lacking constructions of a more restricted abstract level that enable a smooth transformation of non-deterministic choice and composition. The semantic gap between non-deterministic choice and composition and the common constructions of the languages that support non-deterministic behaviour and concurrency is almost to wide to be bridged efficiently. Besides this fact remains also the cost of applying these language constructions freely. The overhead that is introduced will strongly effect the performance of the program and must be taken into account at early design stages.

### 5.4.2.3 CCS in Verification

Verification of CCS agents assumes a relation between the agents and a means to describe assertions of the agents, i.e. some kind of specification of the behaviour of the agents. This section briefly summarizes two important techniques to describe the assertions: by logic formulae and by set of traces.

**Process logic** MILNER and HENNESSY have developed a means to specify the external behaviour of processes in a logic style. This logic $\mathcal{P}L$, often denoted by the term Hennessy-Milner logic [100], is a modal logic. Its formulae are build from the following abstract syntax:

$$F ::= \text{true} \mid \text{false} \mid F \land G \mid F \lor G \mid (a)F \mid [a]F$$

This version is without negation, following [100]. MILNER presents a version with negation in [156]. An informal interpretation of the two modalities are:

- the modality $(a)G$ asserts of an agent $P$ that it is possible for $P$ to do action $a$ and thereby reach a state $Q$ which satisfies formula $G$ (similar to the $\Diamond$ modalities of temporal logic) and

- the modality $[a]G$ asserts of an agent $P$ that if $P$ can perform action $a$ it must reach a state where $G$ holds (similar to the $\Box$ modalities of temporal logic).

LARSEN has given an intuitionistic interpretation of this logic in [135].

Given the logic $\mathcal{P}L$ and the agents of basic CCS, $\mathcal{PE}$ a satisfaction relation $\models$ can be defined that relates the agents to an assertion of $\mathcal{P}L$
under assumption of the semantic SOS model for $\mathcal{PE}$.

$$
P \models \text{true}, \forall P \in \mathcal{PE}
$$

$$
P \models F \land G \text{ if } P \models F \land P \models G
$$

$$
P \models F \lor G \text{ if } P \models F \lor P \models G
$$

$$
P \models \langle a \rangle F, \text{ if } \exists P' \in \mathcal{PE} \cdot P \xrightarrow{a} P' \Rightarrow P' \models F
$$

$$
P \models [a] F, \text{ if } \forall P' \in \mathcal{PE} \cdot P \xrightarrow{a} P' \Rightarrow P' \not\models F
$$

Milner has shown that two agents $P$ and $Q$ are bisimilar if and only if no $\mathcal{PL}$ formula distinguishes between them. Thus $\mathcal{PL}$ offers an alternative way to characterise bisimilarity.

This (very) basic logic can be extended with all kind of derived operators. E.g. the modal logic of Bradfield and Stirling [42], the (extended) modal-$\mu$ calculus is such an extension of the logic $\mathcal{PL}$ that can be used to check assertions of CCS with value passing. The most specific extension is the possibility to define fixed point operators that define greatest and least fixed points of propositions. For model checking in this logic a tableau system is derived that enables the verification of both safety and liveness properties. The logic and the tableau method are incorporated in the Edinburgh CCS toolset [159].

**Process traces** An important observation with respect to the application of these logics is that they are not compositional in the following sense. There is no function $f_i$ such that

$$
P \models F \land Q \models G \Rightarrow P \models Q \models f_i(F, G).
$$

Another drawback of these logics is the way to show that a process does not meet a logic specification. This is done by showing that the process meets the negation of the logic specification, which is not a trivial fact to accomplish. These two phenomena follow from the fact that the calculus that governs the $\models$ relation is based on the structure of the formulae of the logic and not on the structure of the formulae of the agents. This means that the whole agent $P$ must be developed before any assertion over it can be checked. According to these properties of the modal calculus Hoare argues that modal logic is not the appropriate tool in the context of verifying communicating processes. He offers an alternative approach based on two more or less related semantic constructions, the traces and failures of processes. The satisfaction relation is based on defining an assertion as a set of traces or failures of a process and assertion checking proceeds by proving that the behaviour of the process delivers exactly that set of traces or failures. This approach is compositional, that is, there are functions such
5.4. The models of VDM, CCS and TIME

that for $op \in OP$ with $OP$ the set of behaviour operations

$$(P \models F \land Q \models G) \Rightarrow (P \models op Q \models f_{op}(F, G)).$$

Traces form a weaker means to distinguish processes than bisimulation. For example two agents

$$A \triangleq a.(b.0 + c.0)$$

and

$$B \triangleq a.b.0 + a.c.0$$

are equivalent under traces but are not bisimulations. This can be easily shown. Let

$$F \equiv \langle a \rangle(\langle b \rangle \text{true} \land \langle c \rangle \text{true})$$

Then it holds that $A \models F$ and $B \not\models F$, and so $A \not\sim_F B$. On the other hand, let

$$TR = \{[], [a], [b], [a, b], [a, c]\}$$

be a set of traces. Then it holds that $A \models TR$ and $B \models TR$ and thus $A \sim_T RB$.

Both techniques have their advantages and disadvantages, from which some have been demonstrated above. By side-by-side application within the verification process they offer very strong techniques to verify the behavioural properties of communicating agents. Still they lack the power to verify properties of the data elements that are part of the CCS specifications.

5.4.3 Time in software development

The time ingredient of the different techniques and notations used during the various stages of software engineering is strongly influenced by the capabilities of the different techniques. In [163] Mortus captures these differences by looking at the notion of time from three different viewpoints: time as seen by the implementor, time as seen by the verifier and time as seen by the specifier. In general, the implementors use a discrete time notion, the verifiers use both continuous and discrete time and specifiers use, for pragmatic reasons, mostly discrete time.

5.4.3.1 Time and Implementation

For the implementor the only reality is the computer system, the rest of the world, i.e. the environment of the computer system, is a model, which can be sometimes very detailed. The implementor is interested in time
that provides a basis for organizing the execution of programs and may be used in implementation bound theories like scheduling. Kopetz [128] distinguishes four different kinds of time.

- Physical time, where a reference is established by counting cycles of a physical, strictly periodic process (i.e. the ticks of a physical clock). This time is usually metric, as there is the need for a measure to express the distance of one tick from another.

- Logical time, where a reference is established by counting specified significant events during the execution of a program (logical ticks). The time between ticks is not measurable.

- Absolute time, where a reference is established in relation to a global event for a given system (e.g. the origin of absolute time may be the instant of switching a system on). There is only one absolute time reference in the system.

- Relative time, where a reference is established in relation to a local event in the given system. As a result there is usually more than one relative time in a system.

E.g. in distributed systems global time and local time exist side by side and there is in general a difference between the two time notions. Global and local time attributed can be used both for logical and physical time. Usually absolute time is global and relative time local.

5.4.3.2 Time and Verification

This viewpoint is typical for the design stage of software development. Mostly the verifier is interested in all possible executions of a program in time, and concentrates on certain properties of the behaviour of the program like safety and liveness properties and fairness aspects. Time may be reduced to the ordering of events (i.e. logical time) without using any metric at all. Verifier works in general with computational models of programs, and consequently their time notions is related to that of the implementor. In the next section the tools of the verifier, being logics of time, are discussed in more detail.

5.4.3.3 Time and Specification

The specifiers face two major tasks:

- the presentation of time constraints and requirements of the system components and
• matching the different time counting systems that are applied in a system.

Each of these tasks may be described in different ways, using different notions of time. DASARATHY in [62] gives a systematic overview of the various forms of timing constraints. MOTUS ET AL. [164] present a systematic approach for relating a variety of time notions.

5.5 **MOSCA in software development**

The statements made concerning the rôle of VDM and CCS in software development are summerized in the next table (figure 5.3. To avoid a too detailed classification there are only two different forms of capabilities used: weak and strong. The table states the applicability of VDM and

<table>
<thead>
<tr>
<th>Development Topic</th>
<th>Specific product</th>
<th>VDM</th>
<th>CCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>env. model</td>
<td>strong</td>
<td>weak</td>
</tr>
<tr>
<td></td>
<td>beh. model</td>
<td>strong</td>
<td>weak</td>
</tr>
<tr>
<td>Implementation</td>
<td>arch. model</td>
<td>strong</td>
<td>strong</td>
</tr>
<tr>
<td></td>
<td>impl. model</td>
<td>strong</td>
<td>strong</td>
</tr>
<tr>
<td>Verification</td>
<td>static anal.</td>
<td>strong</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dynamic anal.</td>
<td>weak</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: VDM and CCS in the SDA

CCS to data modelling and control modelling to the four different models in the software development activities. A strong capability suggests efficient description facilities for the data or control facets of the specific product, a weak capability suggest the unsuitability of the notation.

VDM's weak capability to specify the control in the environmental and behavioural model is based on the ability to specify control only axiomatically on this level (by pre- and post conditions). This approach leads to overly complex and unreadable specifications for state based systems.

CCS's weakness is clearly the (almost) absent features for data modelling. Its weakness to specify control in the implementation models is clearly caused by the bias towards parallelism in stead of sequential control based on iteration.

The coverage of VDM and CCS in the context of the phases within the software development life cycle are summarized in figure 5.4. The strength
of the coverage in a specific phase is symbolically expressed as the thickness of the area.

It is tempting to suggest that MOSCA inherits all strong points of the two notations, tempting but wrong off course! MOSCA retains the strong concepts of data description of VDM, but unfortunately clearly copies CCS in the case of control abstractions.

Given these last observations I postulate now an answer on the second question stated in the introduction of this chapter.

**Question:** Given the particular view on software development contained in the software development framework, where does MOSCA fit in?

**Answer:** In the context of software development MOSCA is best equipped to be used as specification language to record the behavioural requirements of software components of TSCS's. It offers both data abstraction and control abstraction constructions that enable the specification of environmental and behavioural models.

The timing constructions of MOSCA are best equipped to model the various aspects of time within the implementation models. The timing constructions of MOSCA offer an operational view on time management. In the requirement specification phase, however, the timing aspects are usually not easily expressed in an operational style. More often timing aspects are expressed as sentences in the form of "It will not be the case that during the next X seconds P will happen" or "During its whole lifetime Q will respond with action R within Y clockticks" etc. These requirements have in general a more axiomatic flavour and are best expressed within some form of temporal or timed logic formula over the behaviour of the software component.

As such, the non-behavioural aspects that involve timing, together collectively addressed as the timing constraints of the software components
may better not be expressed in MOSCA. The MOSCA model describes the behaviour, including time-dependent behaviour and forms the basis to discharge the proof obligations induced by the timing constraints.

In chapter 9 I will come back to this conjecture to investigate its truth value in the context of the collected material from the next three chapters.
6.1 Introduction

There are several ways to study the behaviour of MOSCA agents. Two approaches are introduced in the two following sections. The first approach addresses state spaces of agents, based on the $STS$ type. The second approach introduces traces over the $STS$ elements. Both approaches are based on the SOS definition of MOSCA. The closing section shortly summarizes the results on state spaces and traces and offers some observations on other approaches.

6.2 State space analysis

6.2.1 Introduction

State space analysis is a powerful technique to establish safety and liveness properties of MOSCA agents. In particular state space analysis offers

- a means to analyse the termination properties of agents: divergence analysis, termination and deadlock analysis;

- a vehicle to perform reachability analysis, which answers the question whether a specific $STS$ configuration can be reached starting from a given behaviour expression and environment.

State spaces have one important drawback that often disables their usage all together: their possible infinite size. In a recent overview of current advances in distributed computing and communications YEMINI (in [224]) has collected some specific topics concerning the complexity of searching vast state spaces. HOLZMANN presents techniques to search only parts
of the total state space in a heuristic way (scatter searching, [105]) and provides a means to analyse state spaces by tracing ([106]).

Although the findings of the current research in scatter searching techniques and tracing are encouraging I have investigated a different approach to tackle the size problem i.e. reducing the size of the search space by abstracting from aspects that are not involved with the topic of the analysis. E.g. if the analysis topic is the occurrence of deadlock during the lifetime of a process, it may be the case that the phenomenon that represents the deadlock situation in the state space is recognisable not only in the total state space $S$ of the process but also in a reduced state space $S'$ that results from the application of a contraction $\gamma$ on $S$.

In this section different classes of derived state spaces are defined. Each class is derived from the total state space by a certain abstraction transformation. A derived state space class exhibits a specific aspect of the MOSCA behaviour expression for which it is computed. The total state space of a MOSCA behaviour expression is formed by the set of different state transition states of the transition graph of the behaviour expression. This state space is formally defined in the following subsection.

The different classes of state spaces are fixed by two properties: the label set over which transitions can be made and the specific class of behaviour expressions. Let $SS_r$ denote the state space in which the transitions are chosen from set of possible actions $Act \setminus r$. The class $SS_r$ is further divided into four subclasses fixed by certain properties of the $STS$ elements. The first subsection treats the standard state space which forms the starting point of the state space development. Next all derived state spaces are introduced. The next two sections define the three different kind of contractions that are defined on the standard state space. Section 6.2.5 discusses relationships between the different state spaces. Section 6.2.6 discusses a technique to generate (parts) of the defined subspaces explicitly.

In the following sections some additional notation is used.

Let $\mathcal{R}$ be the set of possible restriction sets.

$$\mathcal{R} = \{\varnothing, \alpha, \iota, \epsilon, \alpha\iota, \alpha\epsilon, \iota\epsilon, \alpha\iota\epsilon\}.$$ 

Let $Act_r$ be some restricted action label set with $r \in \mathcal{R}$. Let $as \in Act_r$ and $n = \text{len } as$ then $\xrightarrow{as}$ is a shorthand for

$$\xrightarrow{as(1)}_r \xrightarrow{as(2)}_r \cdots \xrightarrow{as(n)}_r$$

The subscript $r$ is omitted from $\xrightarrow{-}$ and $Act$ if $r = \varnothing$.

The notation $s \xrightarrow{a}_r$ denotes a predicate stating whether a state $s$ has an
6.2. State space analysis

a transition over \( \text{Act}_r \).

The notation \( \xrightarrow{*} \) denotes the reflexive, transitive closure of the transition relation.

6.2.2 The total and derived state spaces

**Definition 6.1** (Rooted transition system) A rooted transition system \( \text{RTS} = (S, TL, R, s_0) \) is defined through a set\(^1\) of nodes \( S \), a set of transition labels \( TL \), a relation \( R \subseteq S \times TL \times S \) and a node \( s_0 \) such that \( s_0 \in S \). \( s_0 \) forms the root of the transition system.

The following three basic functions are defined over a rooted transition system:

\[
\text{First}(s) \triangleq \{ tl | tl \in TL : (s, tl, -) \in R \}
\]

\( \text{First}(s) \) computes the set of actions that \( s \) can perform initially.

\[
\text{Action}(s, a) \triangleq \{ (s, a, s') | s' \in S : (s, a, s') \in R \}
\]

\( \text{Action}(s, a) \) computes the result of doing an \( a \) action on \( s \). The set of all states in the relation \( R \) that can be reached starting from state \( s \) is defined as:

\[
\text{States}(s) \triangleq \{ s' | sR^*s' \}
\]

where \( R^* \) denotes the reflexive transitive closure of \( R \).

**Definition 6.2** The transition system associated with a MOSCA state transition state \((be, \rho)\) is

\[
\text{MTS}((be, \rho)) = (\text{STS}, \text{Act}, \xrightarrow{*}, (be, \rho)).
\]

The total state space \( \text{TSS} \) is defined as

\[
\text{TSS} = \text{States}((be, \rho)).
\]

The derived state spaces \( SS \) are defined by variation of two properties. The first property is a specific shared quality of the \( \text{STS} \) elements that form the basis for the state space states. The second property is the set of action labels that may be used in the transitions. The first property distinguishes between:

\(^1\)In the following all sets are not restricted to finity cardinality, unless it is stated otherwise.
1. full states (\(SS^F_r\)), in which the state space type is equal to the \(STS\) type from the \(MOSCA\) semantics, or the reduced states (\(SS^R_r\)), in which the state space type is limited to behaviour expressions \(be\) and environments \(rho\) from which the value components are deleted, thus reducing \(MOSCA\) to \(TCCS\),

2. the deterministic state space (\(SS^D_r\)), derived from either the full or reduced state spaces by merging all non-deterministic behaviour induced by the choice operator. The non-deterministic standard state space is the default, such that \(SS^F\) is interpreted as the non-deterministic full state space.

**Definition 6.3 (Restricted State spaces)** Let \(\alpha\) denote the set of external actions \(ExtActs\), \(i\) denote the singleton set of the internal action \(\{i\}\), \(\epsilon\) denote the set of all time actions \(TimeAct\).

\[
\begin{align*}
\alpha & = \{a \mid a \in ExtActs\} \\
i & = \{i\} \\
\epsilon & = \{\epsilon(d) \mid d : T\}
\end{align*}
\]

Juxtaposition of two label sets denotations is interpreted as set union. The following derived state spaces are defined:

\(SS_\emptyset, SS_\alpha, SS_i, SS_\epsilon, SS_{\alpha i}, SS_{\alpha \epsilon}, SS_{i\epsilon}, SS_{\alpha i\epsilon}\)

both for full, reduced and deterministic state spaces. In \(SS_r\) the subscript \(r\) denotes the restriction set. The actions (i.e. the labels on the arcs in the transition graph) may be taken from \(Act \setminus r\).

Summing up the state spaces we get the picture in 6.1. State spaces may be finite or infinite. The time component and VDM looseness, when applied in a specification, may easily cause its state space to become (wildly) infinite. The different state spaces offer the possibility to abstract from either VDM looseness, time components or internal actions. Often infinite full state spaces will have finite reduced equivalents. In the following subsections these classes are defined properly, presented by a generating function and discussed by several examples.

### 6.2.3 The state space classes \(SS^F_r\), \(SS^R_r\) and \(SS^D_r\)

**Definition 6.4 (derived state space transition)** Let \(P, Q \in STATE\)-set then there is a state space transition of \(P\) to \(Q\) over action \(s \in Act_r\), denoted as \(P \xrightarrow{s} Q\) iff
Figure 6.1: The total and derived state spaces
\[ \exists p_i \in P \cdot p_i \xrightarrow{s} \land \]

let \( P_s = \{ p_i \mid p_i \in P \land p_i \xrightarrow{s} \} \) in

\( Q = r\text{-}closure(q_j) \) such that \( q_j \in Action(p_j, s) \land p_j \in P_s \)

where the \( r\text{-}closure \) is defined inductively as:

\[
\begin{align*}
    r\text{-}closure(ss) &= \{ \} & \text{if } ss \text{ is empty,} \\
    r\text{-}closure(ss) &= ss \cup r\text{-}closure(nss) & \text{otherwise}
\end{align*}
\]

such that

\[
nss = \bigcup\{Action(st, rs) \mid st \in ss, rs \in r\}
\]

The \( r\text{-}closure \) computes the reflexive transitive closure of the relation \( \rightarrow_r \).

\[\square\]

**Definition 6.5 (Full state space \( SS_r^F \))** Let \( STATE \) be the domain of all possible states of the state space:

\[
STATE = STS\text{-}set
\]

where \( STS \) is the type of state transition states, defined in definition 4.2. The full state space \( (SS^F) \) of a behaviour expression \( be \) and some execution environment \( \rho \) consists of sets of powersets \( st \in STS\text{-}set \) of state transition states. A state space element \( S \in STATE\text{-}set \) is in \( SS_r^F((be, \rho)) \) iff there is a sequence of state space transitions \( ts \) such that

\[
r\text{-}closure((be, \rho)) \xrightarrow{ts} r S
\]

or more formally

\[
SS_r^F((be, \rho)) = \{ SS \mid SS: STATE \cdot \exists ts: (Act \setminus r)^* \cdot (be, \rho) \xrightarrow{ts} r SS \}
\]

\[\square\]

The state spaces \( SS_r^F \) are called full as they are generated by all possible transitions in \( Act \setminus r \) over the full MOSCA behaviour expression \( be \) executing in an environment \( \rho \).

The reduced state spaces \( SS_r^R \) abstract away from the value environment component of a sts element and from all value constructions in the behaviour expression, thus reducing MOSCA to TCCS without value passing. Each node in the \( rss \) represents a class of nodes from the \( TSS \). The reduced state space is defined in a similar way as the full complete state space.

**Definition 6.6 (Reduced state space \( SS_r^R \))** The reduced state space \( (SS^R) \) of a behaviour expression \( be \) and some execution environment \( \rho \) consists of sets of powersets \( st \in STS\text{-}set \) of state transition states. A state
space element $S \in STATE$-set is in $SS^R_r((be, \rho))$ iff there is a sequence of state space transitions $ts$ such that

$$r\text{-closure}(SVP((be, \rho))) \leftrightarrow^{ts}_r S$$

or more formally

$$SS^R_r((be, \rho)) = \{SS \mid SS : STATE \cdot \exists ts : (Act \setminus r)^* \cdot SVP((be, \rho)) \leftrightarrow^{ts}_r SS\}$$

where $SVP(sts)$ strips the value parts $Genv$ and $Cenv$ from the environment $\rho$. 

The deterministic state spaces $SS^D_r$ are defined through an adaptation of the definition of the derived state space transition.

**Definition 6.7 (deterministic state space transition)** Let $P, Q \in STATE$-set then there is a deterministic state space transition of $P$ to $Q$ over action $s \in Act_r$, denoted as $P \xrightarrow{\sigma_r} Q$ iff

$$\exists p_i \in P \cdot p_i \xrightarrow{s} \land$$

let $P_s = \{p_i \mid p_i \in P \land p_i \xrightarrow{s}_r\} \text{ in }$ $Q = \bigcup\{r\text{-closure}(q_j) \mid q_j \in Action(p_j, s) \mid p_j \in P_s\}$

In the deterministic state space all non-determinism is deleted and replaced by merged deterministic behaviour.

**Example 6.1** The $TSS$ and $SS^R$ of agent $P$ as defined in specification 6.1 are given in figure 6.3. These two state spaces show the effect of merging.
6.2.4 The state spaces induced by restriction

6.2.4.1 The complete state spaces

The complete state spaces FCSS and RCSS are defined as:

\[
\begin{align*}
FCSS &= SS_F^0 \\
RCSS &= SS_R^0 \\
FDCSS &= SS_{FD}^0 \\
RDCSS &= SS_{RD}^0
\end{align*}
\]

Example 6.2 (FCSS and RCSS) Consider the agent Ticker. The enfolded full complete state space of Ticker \( \langle 3 \rangle \) is depicted in figure 6.4. Each state on the central horizontal line represents a ticker \( \langle i \rangle \) state. All states above the central horizontal line represent Down \( \langle p, q \rangle \) states and the other states represent Up \( \langle k, l \rangle \) states. All diagonal transitions are
6.2. State space analysis

<table>
<thead>
<tr>
<th>6.2 Ticker</th>
</tr>
</thead>
</table>

\[
\text{Ticker} \ (n) \triangleq \begin{cases} 
\text{if } n = 0 \\
\text{null} \\
\text{else let } t: \mathbb{N} \text{ be s.t. } t \leq n \text{ in} \\
\text{if } t \mod 2 \\
\text{then } \text{Up}(t, n + t) \\
\text{else } \text{Down}(t, n - t) \\
\end{cases}
\]

\[
\text{Up} \ (t, n) \triangleq \begin{cases} 
\text{if } t > 0 \\
\text{then up } \odot \ \text{Up} \ (t + 1, n) \\
\text{else } \text{Ticker} \ (n) \\
\end{cases}
\]

\[
\text{Down} \ (t, n) \triangleq \begin{cases} 
\text{if } n > 0 \\
\text{then down } \odot \ \text{Down} \ (t - 1, n) \\
\text{else } \text{Ticker} \ (n) \\
\end{cases}
\]

\[t\] transitions representing the let binding activities. All arrows directed downwards represent down transitions, the upwards directed arrows are up transitions. The root of the graph is the ticker \(3\) state. The graph is enfolded completely up to ticker \(6\). There is only one final state, ticker \(0\). Each ticker \(i\) state has \(i + 1\) \(t\) transitions each resulting in either a Up or a Down state. The full graph is clearly infinite.

- **Ticker** may diverge by selecting in each let the value \(t = 0\).

- The behaviour of Ticker may be infinite by selecting in each let an even value for \(t\).

- The behaviour of Ticker may be finite by selecting e.g. the largest possible value for \(t\) if \(n\) is odd.

The reduced complete state space is depicted in figure 6.5. The full state space is collapsed into a three state graph. All states on the horizontal line are collapsed into state 0. All states above the horizontal line into state 2 and all below the line into state 1. The start state of the full state space is mapped on state 0, the final state remains state 0.

The reduced complete state space retains the original properties of the full complete state space with respect to termination. It still reflects the possibility of divergence that was originally present in the full complete state space. It reflects the property of infinite behaviour of the full complete state space by switching from state 2 to 1 or vice versa through internal
actions and it contains a final state, equivalent with the final state of the full complete state space. These observed facts hold in general for each pair of full and reduced state spaces and is proved formally in section 6.2.5.

The reduced state space reflects what kind of behavioural progress can be expected from an agent, where the full state space presents specifically how the progress proceeds.

6.2.4.2 The observational state spaces

The observational state spaces FOSS and ROSS are defined as:

\[
\begin{align*}
\text{FOSS} & = SS_{\epsilon}^P \\
\text{ROSS} & = SS_{\epsilon}^R \\
\text{RDOSS} & = SS_{\epsilon}^{PD} \\
\text{RDOSS} & = SS_{\epsilon}^{RD}
\end{align*}
\]
6.2. State space analysis

![Diagram of state transitions]

Figure 6.5: Reduced complete graph of ticker (3)

The observational state spaces 
abstract from all internal actions and time actions. They reveal the development of a given MOSCA behaviour expression in terms of observational actions only. Internal actions and time actions are not observable. The agent $iDiverge$, defined as

\[ iDiverge \]

\[ \text{state } val : \mathbb{N} \]

\[ iDiverge \triangleq \sigma(val := val + 1) \circ iDiverge \]

is continuously incrementing its state value $val$ without ever producing an observable action. Its observational state space is by definition empty. Similarly the agent $eDiverge$ defined as

\[ eDiverge \triangleq \text{idle}3 \circ eDiverge \]

which is continuously idling 3 time units, never produces any observable action. Again the observational state space is by definition empty.

**Example 6.3** The Ticker agent is extended to signal its termination by synchronization via a `stop` action before transforming into the null agent. The agent `Count` registers all ongoing ticks, either up or down. Let the agent $TC$ be defined as:

\[ TC = (Ticker (3) \mid Count) \setminus \{up, down\}. \]

$TC$ generates an infinite complete state space isomorphic to the complete state space of Ticker (3).

The reduced complete state space of the composition is finite but abstracts away the essence of the computational effect of the composition completely. \qed
6.3 Count agent

\[
\begin{align*}
\text{Count} & \triangleq \text{Counting} \langle 0, 0 \rangle \\
\text{Counting} \langle u, d \rangle & \triangleq \\
& \text{up} \odot \text{Counting} \langle u + 1, d \rangle \oplus \\
& \text{down} \odot \text{Counting} \langle u, d + 1 \rangle \oplus \\
& \text{stop} \odot \text{result}(u, d) \odot \text{null}
\end{align*}
\]

6.2.4.3 The timeless state spaces

The timeless state spaces $FTSS$ and $RTSS$ are defined as:

\[
\begin{align*}
FTSS & = SS_F^e \\
RTSS & = SS_R^e \\
RDTSS & = SS_{PD}^e \\
RDTSS & = SS_{RD}^e
\end{align*}
\]

The timeless state spaces abstract from all idle actions. They reveal the development of a given MOSCA behaviour expression in terms of external and internal actions only.

Example 6.4 In this setting the time domain is discrete: $T = \mathbb{N}$. Let $IOC$ be defined as the composition of the clock agent and IO agent as defined in specification 6.4.

\[
IOC = \text{clock} \langle 0 \rangle \mid IO \langle 0 \rangle
\]

The total state space of $IOC$ is infinite. $IOC$ offers the waiting time on input actions and accumulates the total waiting time. The partially enfolded state state space of $IOC$ is given in figure 6.6. The open nodes in the graph symbolize $IO \langle m \rangle \mid clock \langle n \rangle$ states. The filled nodes symbolize states in which the execution has evolved to

\[
\text{output}(t) \odot IO \langle m + t \rangle \mid clock \langle n \rangle
\]

The horizontal transitions are the $e(1)$ transitions of the composition, the transitions upwards from open nodes model input actions, upwards or downwards from closed nodes they model output actions. $SS_F^e(IOC)$ and
6.2. State space analysis

\[
\text{clock } \langle T \rangle \quad \text{[6.4 IO and clock]}
\]
\[
\text{clock } \langle t \rangle \triangleq \text{idle}_1 \odot \text{clock } \langle t + 1 \rangle
\]

\[
\text{IO } \langle N \rangle
\]
\[
\text{ports syn input}
\]
\[
\overset{\text{out}}{\text{output}} : N
\]

\[
\text{IO } \langle dt \rangle \triangleq \text{input}_t \odot \text{output}(t) \odot \text{IO } \langle dt + t \rangle
\]

\(SS^D_\epsilon(IOC)\) are depicted in figure 6.8. \(SS^F_\epsilon(IOC)\) is still infinite and due to the merging of all \(\epsilon\) actions it has become non-deterministic as well. Each node that models \(\text{IO } \langle m \rangle\) has an infinite number of outgoing \text{input} actions to all states that model \(\text{output}(t) \odot \text{IO } \langle l + t \rangle \mid \text{clock } \langle n \rangle\) such that \(l > m\). \(SS^D_\epsilon(IOC)\) is reduced to a simple two state graph. \(\square\)

6.2.5 Properties of state spaces

As stated in the introduction section the state space analysis is mostly concerned with either termination properties or reachability issues. In this section I will show which properties are invariant over the different contractions defined in the last two subsections. First some notation is introduced.

Each of the derived state spaces can be viewed as a contraction that transforms a given rooted transition system \(RTS\) into another rooted transition system \(RTS'\).

**Definition 6.8 (contraction)** A contraction is a mapping over rooted transition systems. A contraction \(C(\mu S, \mu TL, \mu R, \mu S_0)\) is a 4-tuple of mappings that map the 4 parts of a rooted transition system into the parts that form a new rooted transition system, the contracted transition system. \(\square\)

The derived state spaces are products of a family of contractions on

\((STS, \text{Act}, \rightarrow, \text{sts})\)

where:

1. \(\mu STS\) is the powerset operation on the \(STS\) domain,
2. given the restriction set \(r \mu Act\) is defined as: \(\{a \mid a \in Act \setminus r\}\),
3. \(\mu \rightarrow\) is a map that transforms the given relation \(\rightarrow\) into either \(\leftrightarrow\) or \(\Rightarrow\),
4. \( \mu s_0 \) is either \( r\text{-}closure(s_0) \) or \( r\text{-}closure(SVP(s_0)) \).

Let \( C_{XY} \) denote the contraction that results in \( SS_{XY} \).

**Definition 6.9 (space mapping function)** A contraction \( C_{XY} \) realizes a function \( S_{XY} \) between the state space types of the domain and co-domain rooted transition systems. Let \( \langle STS, Act, \rightarrow, sts \rangle \) be the domain rooted transition system for \( C_{XY} \), then

\[
S_{XY}(s) \triangleq \\
\text{let } rss = \{ r\text{-}closure(s_i) \mid s_i \in States(s_0) \} \text{ in } \\
\{ rs \mid rs \in rss \land s \in rs \}
\]

pre \( s \in States(sts) \)

that is, \( S_{XY}(s) \) computes the set of all closures from which \( s \) is a member. \( \Box \)

In the following the contractions are symbolically denoted by the same denotations that were used to characterize the state spaces.
6.2. State space analysis

Figure 6.7: Partially enfolded $SS^F_\ell$ graph of IOC

Figure 6.8: $SS^D_\ell$ graph of IOC

Assuming the basic functions $States$, $r$-closure, First and Action to be defined then the following predicates express some important properties of the RTS:

- **(termination property)**

  $T : S \rightarrow B$

  $T(s) \triangleq \forall l \in LS \cdot (\neg \exists s' \in S \cdot (s' \neq s \land s \xrightarrow{a} s'))$

- **(strong set termination)**

  $T : S$-set $\rightarrow B$

  $T(ss) \triangleq \forall s \in ss \cdot T(s)$

- **(weak set termination)**

  $\hat{T} : S$-set $\rightarrow B$

  $\hat{T}(ss) \triangleq \exists s \in ss \cdot T(s)$

- **(divergence property)**

  $D(s) \triangleq \forall ic \in \nu$-closure$(s) \cdot (First(ic) = \{\ell\})$
• (strong set divergence)
\[ D : \text{S-set} \rightarrow \mathbb{B} \]
\[ D(ss) \triangleq \forall s \in ss \cdot D(s) \]

• (weak set divergence)
\[ \tilde{D} : \text{S-set} \rightarrow \mathbb{B} \]
\[ \tilde{D}(ss) \triangleq \exists s \in ss \cdot D(s) \]

• (reachability)
\[ R(s) \triangleq s \in \text{States}(s_0) \]

The first predicate tests whether a state \( s \) is terminal, i.e. whether it has any outgoing transitions at all. Termination is defined over sets of states in both strong and weak forms. The second predicate tests whether a given state \( s \) diverges, i.e. whether the state has only \( \epsilon \) transitions in its closure. Again there are weak and strong versions for sets of states. The third predicate tests whether a state \( s \) is reachable, i.e. whether it is in the state space of the rooted transition system. In the context of more than one RTS the standard functions \textit{States}, \textit{r-closure}, \textit{First} and \textit{Action} and the three predicates are indexed with the label of the RTS for which they apply.

**Proposition 6.1** Let \( DOM = (STS, Act, \rightarrow, r_0) \) be the rooted transition system for \( \lambda_0 = (bc, \rho) \). Let \( C \) be any of the defined contractions, resulting in a rooted transition system \( RNG = (STS\text{-set}, Act \setminus r, DR, d_{s_0}) \). Let \( S \) be the associated state mapping function. Then the following holds:

1. (Preservation of space structure)
\[ ((s, a, s') \in \rightarrow \land a \notin r) \Rightarrow \]
\[ (\forall ds \in S(s) : (\forall ds' \in S(s') : ((ds, a, ds') \in DR))) \]

2. (Preservation of termination)
\[ (a) \forall s \in States_{DOM}(s_0) : (T_{DOM}(s) \Rightarrow \forall ds \in S(s) \cdot \tilde{T}_{RNG}(ds)) \]
\[ (b) \forall ds \in States_{RNG}(d_{s_0}) : ((\forall s \in ds \cdot T_{DOM}(s)) \Rightarrow T_{RNG}(ds)) \]
3. (Preservation of divergence) If \( \iota \notin r \) then

(a) \( \forall s \in Stages_{DOM}(s_0) \cdot (D_{DOM}(s) \Rightarrow \forall ds \in S(s) \cdot \bar{D}_{RNG}(ds)) \)

(b) \( \forall ds \in Stages_{RNG}(ds_0) \cdot ((\forall s \in ds \cdot D_{DOM}(s)) \Rightarrow D_{RNG}(ds)) \)

4. (Preservation of reachability)

\( \forall s \in S \cdot (R_{DOM}(s, s_0) \Rightarrow \forall ds \in S(s) \cdot R_{RNG}(ds, ds_0)) \)

Proof:

1. Two cases occur: (a) \( DR = \leftrightarrow \) or (b) \( DR = \rightarrow \). For both cases the same argumentation is valid. The proposition is proved by induction over the structure of \( R \).

   (a) \( DR = \leftrightarrow \). By definition it holds that \( d_{s_0} = r\text{-closure}_{RNG}(s_0) \) or \( d_{s_0} = r\text{-closure}_{RNG}(SVP(s_0)) \). For both cases the same arguments hold. First the induction basis is established. If \( s_0 \xrightarrow{\iota} a \) and \( a \notin r \) then according to definition 6.4 \( d_{s_0} \xrightarrow{\iota} r\text{-closure}(s_a) \). Now consider \( s \xrightarrow{a} s' \). There must be a state \( x \in Stages_{DOM}(s_0) \) such that there is a sequence of transitions \( as \in (Act \setminus r)^* \) such that \( s_0 \xrightarrow{as} x \), \( s \in r\text{-closure}(x) \) and \( x \xrightarrow{rs} s \) such that \( rs \in r^* \). Now \( r\text{-closure}(x) \in S(s) \). Let \( X = r\text{-closure}(x) \). The state \( X \) must be in \( Stages_{RNG}(d_{s_0}) \) again due to definition 6.4. Similarly there must be a state \( p \) in \( Stages_{DOM}(s_0) \) such that \( s_0 \xrightarrow{as'} p \), \( p \xrightarrow{rs'} s' \) and \( s' \in r\text{-closure}(p) \). Let \( P = r\text{-closure}(p) \). Again it must hold that \( P \in Stages_{RNG}(d_{s_0}) \). The state \( X \) in \( Stages_{RNG}(d_{s_0}) \) must have a \( a \) transition to \( P \), according to definition 6.4, so \( X \xrightarrow{a} P \) and \( X \in S(s) \) and \( P \in S(s') \). As there has been no assumption on the choice of \( x \) and \( p \) it must be the case that:

\[
\forall x, p \in Stages_{DOM}(s_0) \cdot \\
\text{let } as, as' \in (Act \setminus r)^* \\
rs, rs' \in r^* \\
X = r\text{-closure}(x), P = r\text{-closure}(p) \text{ in } \\
((s_0 \xrightarrow{as} x \xrightarrow{rs} s \wedge s_0 \xrightarrow{as'} p \xrightarrow{rs'} s') \Rightarrow (s \xrightarrow{a} s' \Rightarrow X \xrightarrow{a} P))
\]

Rewriting of this last result gives the claim in the proposition. The applied argumentation is visualised in figure 6.9.

2. First case (a) is considered. The property of termination of state \( s \) is characterized by the inability of doing any transition, i.e. \( s \xrightarrow{a} \) for any
Figure 6.9: Preservation of space structure

\( a \in Act \). The \( r\)-\textit{closure} of a state \( s \) results in a set of states \( ss \) which may take any transition of the union of all \( \rightarrow \) descendants. In a union the weak termination property is inherited disjointly from any of its members. So if \( s_{ij} = s_{i} \cup s_{j} \) and either \( T(s_{i}) \) or \( T(s_{j}) \) then \( T(s_{ij}) \). Now let \( x \) be in \( States_{DOM}(s_{0}) \) such that \( as \in (Act \setminus r)^{\ast} \) and \( rs \in r^{\ast} \). Then \( s \in r\text{-}closure(x) \).

Let \( X = r\text{-}closure(x) \). Again \( X \) must be in \( States_{RNG}(d_{s_{0}}) \). So it must hold that

\[
X \in S(s) \land T_{RNG}(X),
\]

and again the proposition is proved first by quantification over \( S(s) \) and secondly by quantification over \( States_{DOM}(s_{0}) \).

Case (b) states that the strong termination property holds for \( X \) only iff all member states of \( X \) have the termination property. This fact follows directly from the definitions of the predicate \( T \).

3. The same holds for case (a) of the divergence property:

\[
D(s_{i}) \lor D(s_{j}) \Rightarrow \bar{D}(s_{ij})
\]

that is, if \( s_{i} \) must diverge or \( s_{j} \) must diverge, then \( S_{ij} \) may diverge. which is simply proved by induction over \( \rightarrow \) and definition 6.4. Again let \( x \) be in \( States_{DOM}(s_{0}) \) such that \( s_{0} \xrightarrow{as} x \xrightarrow{rs} s \) and \( as \in (Act \setminus r)^{\ast} \) and \( rs \in r^{\ast} \). Then \( D(r\text{-}closure(x)) \) must hold as \( s \in r\text{-}closure(x) \).

Case (b) is again a direct result of the definition of predicate \( D \).

4. This property is proved using the same argument as in the other cases. Assume the premise \( R_{DOM}(s, s_{0}) \). Let \( x \) be in \( States_{DOM}(s_{0}) \) such that \( as \in (Act \setminus r)^{\ast} \) and \( rs \in r^{\ast} \). Then \( s \in r\text{-}closure(x) \). Let \( X = r\text{-}closure(x) \). \( X \) is in \( S(s) \) and by induction over the structure of \( \xrightarrow{e} \) or \( \xrightarrow{b} \) it follows that \( X \in States_{RNG} \) and subsequently \( R_{RNG}(X, d_{s_{0}}) \).
6.2. State space analysis

By proposition 6.1 it has become possible to search the derived spaces for the presence of desirable properties. E.g. if the total state space is suspected to hold a state that diverges then some specific derived state must hold a state that either may or must diverge. The examples so far have shown that the derived state space must be chosen with care in order to retain useful information with respect to the ongoing computation. But to demonstrate to existence of either of the special properties any derived state space may serve, and hence, the most limited space can be selected.

6.2.6 State space generation

Although I have shown that derived state spaces retain important properties with respect to termination, divergence and reachability, the presentation until so far has been devoted purely to theory and has been (although rigorous) basically informal. The theory is built on a pure naive set theory such as axiomized by e.g. Zermelo-Fraenkel ([73]), in which infinite sets are easily dealt with. The theory provides a proof technique that can be used, in combination with the proof system induced by the formal semantics to establish the validity of properties of MOSCA behaviour expressions in the context of an associated environment. The proofs may become lengthy and complex due to the inherent complexity of the formal semantics of MOSCA and as such the theory of derived state spaces may not provide a practical tool at all. It would be far more useful if the state spaces could be generated extensively and successively be inspected for states with certain properties. But how does one generate an infinite state space completely in finite time and space? It seems all but inevitable that the problem that was stated in the introduction has not been solved in a realistic manner by providing a proof system and a proof technique that still works with infinite state spaces.

To get a better understanding of the problem related to state space generation let's inspect a generating function for the full state spaces $SS_F$, specified in VDM-SL, and as such limited to handle finite sets.

6.2.6.1 Generation of the full state spaces

In the generating function below a full state space is modelled by three components that together contain the transition graph:

1. $ns$, a sequence of nodes,
2. ass, a sequence of action sequences that defines for each node $n(i)$ in $ns$ the action sequence $ass(i)$ that contains the sequence of transitions along the path from the root up to the node $n(i)$ at the time the node came into existence during the construction of the transition graph; the paths defined through these action sequences form as such a spanning tree of the transition graph,

3. $fsts$, a transition step set that holds all transitions of the whole graph. A transition is modelled as a triple, $(from, act, to)$ where $from$ and $to$ are indices in $ns$.

$$STATE = STS\text{-set}$$

$$NS = STATE^*$$

$$AS = Act^*$$

$$ASS = AS^*$$

$$FSTS = (N_1 \times Act \times N_1)\text{-set}$$

$$FSS :: ns : NS$$
$$\qquad ass : ASS$$
$$\qquad fsts : FSTS$$

The computation of the full state space proceeds as follows. Starting with the initial graph $g = g_0$ for all nodes in the graph all transitions are inspected and added to the graph $g$, giving $g'$. This process is repeated until $g' = g$, which will only hold if the complete state space is finite. For infinite state spaces the computation can be stopped after computing the closure up to a given number of nodes in the graph. Although complete analysis is not possible for these cases, it is often important to have a means to study evolving partial state spaces. To mark the uncompleteness of the closure, the node type is refined into

$$STATUS = CLOSED | OPEN$$

$$NS = STATE \times STATUS^*$$

The cardinality of the node set of the graph can be constrained by an argument to the generating function.

Initially $g_0$ is equal to the sts for which the state space must be computed. Function $FSSClosure$ performs the closure with the optimization that only new nodes of $g'$ are processed. It is assumed that $r$ is given. The auxiliary functions $NewNodes$ filters new nodes from the set of transitions,
6.2. State space analysis

\[ FSS_r : \text{STATE} \times \mathbb{N} \rightarrow \text{FSS} \]

\[ FSS_r(sts, max) \triangleq FSS\text{Closure}(1, \text{mk-FSS}([r\text{-closure(sts)}], [], []), max) \]

\[ FSS\text{Closure} : \mathbb{N}_1 \times \text{FSS} \times \mathbb{N} \rightarrow \text{FSS} \]

\[ FSS\text{Closure}(i, fss, max) \triangleq \]

let \( \text{mk-FSS(ns, ass, fsts)} = fss \) in

if \( i > \text{len ns} \lor i >= max \) then \( fss \)

else let \( \text{firstacts} = \circ \text{First}_r(ns(i)) \) in

let \( \text{transitions} = \bigcup \{(\text{let (is, card)} = \circ \text{Action}_r(ns(i), a) \text{ in}
\]

if \( \text{card} = \bot \)

then \( \{(ns(i), a, \{\})\} \)

else \( \{a \in \text{firstacts}\} \) in

let \( \text{nodes} = \text{NewNodes(transitions, fss)} \) in

let \( fss' = \text{MarkClosed(fss, i)} \) in

\[ \text{FssClosure(}
\]

\[ i + 1,
\]

\[ \text{AddTrans}(i, \text{transitions}, \text{AddNodes(nodes, fss'))},
\]

\[ \text{max} \)

\]
which are subsequently added to the node set by \textit{AddNodes}. \textit{AddTrans} adds the new transitions to the \textit{fsts} component of \textit{fss}.

Although the presented specification provides a total generating function, \textit{FssClosure} is in fact a partial function. The partiality may be caused by two different sources: the set bound to \textit{transitions} defined in the third let may try to compute an infinite set, as such causing infinite (breadth) recursion over the graph whilst the sequence of state transitions may grow infinitely, without being cyclic as such causing infinite (depth) recursion over the graph. The latter source of infinite computation can be inhibited by choosing a specific value for the \textit{max} argument. The first source of infinite recursion may be inhibited by restricting the $\circ$\textit{Action} function to produce only sets restricted to hold a certain maximum number of transitions. Whenever this number is exceeded, the result of $\circ$\textit{Action} is replaced by the empty set, thus ignoring possible infinity at all. In a more realistic setting it should be notified somehow which state transition closure is ignored.

### 6.3 Trace analysis

#### 6.3.1 Introduction

In this section timed traces of \textsc{mosca} agents are introduced. They form a means to verify e.g. properties of \textsc{mosca} agents that may be expressed as a history of external observable actions.

Traces were introduced by \textsc{hoare} in [103]. They form the basis for the definition of trace semantics, or trace equivalence, again introduced by \textsc{hoare}. Trace semantics is the weakest form of semantics over processes. A comparative study by \textsc{van glabbeek} relates trace semantics with other forms of semantics in [82]. Traces reflect the \textit{external behaviour} of an agent, without revealing the internal actions of the agent. Although traces can not help to distinguish live agents from dead-locked agents (see e.g. [103]), they offer a means to relate external behaviour of different agents.

Traces can be used to give meaning to the satisfaction relation between a concrete specification and its abstract description, or between an implementation and its specification. (see e.g. \textsc{hoare}'s usage of traces in [103]). The satisfaction relation

$$ P \models S $$

(pronounced as \textit{P satisfies S}) between a program and its specification can be viewed in this context as a relation between \textsc{mosca} specifications (the
programs) and a set of traces (the specification). As such, traces form a strong weapon in proof techniques.

The intuition of a trace of a CCS process is defined in section 2.3.3. A trace of a process consists of a finite sequence of actions, which the process is capable to perform. The total behaviour may then be captured by a (possible infinite) set of traces.

Following the presentation of state space analysis the traces could also be defined for particular properties of the observed behaviour of a process. An observed action involves:

1. its label, that determines the specific action,
2. its communicated value, either being an input or output value, and
3. its moment in time the action was observed.

From this list three sorts of traces can be distinguished:

- $TR^a$, traces that register the action labels, i.e. the standard CCS traces,
- $TR^v$, traces that register the communicated values, and
- $TR^t$, traces that register the moments in time on which an action is observed.

Again these three kinds of traces could be merged thus realizing 7 classes of traces (figure 6.10). $TR^a$ corresponds to the standard intuition of traces.

In the next section I have defined $TR^{a,v}$, the timed traces, in which the basic element in a trace, the action, is extended to hold a time stamp, that records the moment in time the action was taken in relation to the previous action in the trace. The definition of $TR^{a,v,t}$ is straightforward and is easily obtainable from the definition of $TR^{a,v}$. The other trace classes do not seem to provide useful information.

Section 6.3.3 offers some observations on the generation of trace sets. Section 6.3.4 addresses traces for cyclic processes. Finally section 6.3.5 treats the application of trace sets in analysis.

6.3.2 Finite Timed traces

In MOSCA specifications not only the actions play a role, but also the moment in time these actions were performed. The notion of a trace is here refined into the following form. A timed trace of an agent $A$ executing
in environment $\rho$ is the sequence of tuples that forms a history of the behaviour of the agent $A$ up to certain moment in time. As such traces are finite. The tuples are built from external actions and the quantity of time that passed between subsequent actions. The domain of timed traces is defined formally as follows.

$$TTRACE\_ITEM = ExtActs \times TIME$$

$$TTRACE = TTRACE\_ITEM^*$$

where $ExtActs$ is the syntactic domain of all external action labels and $TIME$ is the semantic domain of time value denotations.

The domain of traces is a sequence domain. In the following presentation of the operational characterization of $TR^{\alpha\tau}$ I assume the standard operations on sequences and sets. The notation will be identical to the notation of VDM-SL, but just as in the state space case the sets and sequences are not limited to finite length and cardinality.

The definition of the function that computes sets of traces employs the next functions over traces and sets of traces.

- The function `$\preceq$' acts as restriction on traces and sets of traces in the following way:
6.3. Trace analysis

\[ \triangleleft : TTRACE \times TTRACE\_ITEM\text{-set} \rightarrow TTRACE \]

\[ \triangleleft (t, rset) \triangleq \]

if \( \text{hd } t \in rset \)
then \([\,]\)
else \( \text{hd } t \bowtie t \triangleleft rset \)

\[ \triangleleft : TTRACE\_set \times TTRACE\_ITEM\_set \rightarrow TTRACE\_set \]

\[ \triangleleft (ts, rset) \triangleq \text{let } t \in ts \text{ in} \]
\[ \{ t \triangleleft rset \} \cup (ts \setminus \{ t \}) \triangleleft rset \]

- The function \( \bowtie \) computes the interleaving of traces and sets of traces. First a predicate \( \text{interleaves} \) is defined that checks if a given trace is an interleaving of two other given traces.

\( \text{interleaves} : TTRACE \times TTRACE \times TTRACE \rightarrow \mathbb{B} \)

\[ \text{interleaves}(i, x, y) \triangleq \]

if \( i = [] \)
then \( (x = []) \land (y = []) \)
else \( (x \neq [] \land \text{hd } i = \text{hd } x \Rightarrow \text{interleaves}(\text{tl } i, \text{tl } x, y)) \lor \)
\( (y \neq [] \land \text{hd } i = \text{hd } y \Rightarrow \text{interleaves}(\text{tl } i, x, \text{tl } y)) \)

\( \bowtie (t, u: TTRACE) r: TTRACE\_set \)
post \( i \in r \land \text{interleaves}(i, t, u) \)

\( \bowtie (ts, us: TTRACE\_set) rs: TTRACE\_set \)
post \( ts \subseteq rs \land us \subseteq rs \land \)
\( \forall t \in ts \land \forall u \in us \land (t \bowtie u) \subseteq rs \)

- The function \( \upharpoonright \) functions as renaming operator. It takes a trace or set of traces and a set of renaming maplets.

\[ \upharpoonright : TTRACE \times (\text{ExtActs} \xrightarrow{m} \text{ExtActs}) \rightarrow TTRACE \]

\[ \upharpoonright (t, rmap) \triangleq \]

let \( (a, dt) = \text{hd } t \text{ in} \)
if \( a \in \text{dom } rmap \)
then \( (rmap(a), dt) \bowtie (\text{tl } t \upharpoonright rmap) \)
else \( \text{hd } t \bowtie (\text{tl } t \upharpoonright rmap) \)
\[ \triangleright : TTRACE\text{-}set \times (ExtActs \xrightarrow{m} ExtActs) \to TTRACE\text{-}set \]

\[ \triangleright (ts, rmap) \triangleq \text{let } t \in ts \text{ in } \{ t \triangleright rmap \} \cup (ts \setminus \{ t \}) \triangleright rmap \]

Traces can be recorded in two ways: explicit by enumerating all elements in a trace, or implicit by sequence comprehension. Let us look at some examples of traces before the timed traces of MOSCA agents are defined formally.

Example 6.5 Figure 6.11 presents the set of timed traces of some simple agent expressions.

<table>
<thead>
<tr>
<th>Agent Expression</th>
<th>Finite Timed Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>([\text{}])</td>
<td>{ [[{a}, t]</td>
</tr>
<tr>
<td>(a \odot)</td>
<td>{ [[{a}, t]</td>
</tr>
<tr>
<td>((a \odot \text{null}) \oplus (b \odot \text{null}))</td>
<td>{ [[{a}, t]</td>
</tr>
<tr>
<td>((a \odot \text{null}) \mid (b \odot \text{null}))</td>
<td>{ [[{a}, t]</td>
</tr>
</tbody>
</table>

Figure 6.11: Finite Timed Traces

The first example shows that null agents can not be involved in any actions. The only trace for a null agent is the empty trace. The second example presents a synchronization prefix construction. Now the set of traces consists of the empty trace, reflecting the fact that the agent may never perform the synchronization on \(a\), and of all traces of length 1 that record the moment in time the action was performed. Even for such a basic agent construction the set of traces is infinite. The next two examples display the effect of choice and composition on the traces. The choice operation merges the sets of traces of its operands. The composition operation interleaves the traces of its operands.

\[\square\]
6.3.3 Some remarks concerning trace generation

A central concept in the definition of the timed traces is the idling concept. The set of traces of the second expression in the example is due to the idling abilities of the prefix construction. The expression $a \circ P$ may idle as long as forever. Only when the environment offers a synchronization partner for the prefix expression, the action may take place. In this section I will explain the properties of a function $TR$ with the following signature:

$$TR : STS \times TIME \rightarrow TTRACE\text{-set}$$

and the intuition that $TR((bexpr, \rho), i)$ computes the traces of the agent $bexpr$ executing in environment $\rho$ which has been idling for $i$ time units.

The effect of the idling prefix $idle (d) \circ P$ is now simply expressible as:

$$TR((idle (d) \circ P, \rho), i) \triangleq TR((P, \rho), i + VS[d](\rho))$$

The effect of idling on the other prefixes is addressed next. The effect of a synchronization prefix on the evolving traces is best expressed as a VDM-SL function.

$$Tr : STS \times TIME \rightarrow TTRACE\text{-set}$$

$$Tr((mk-Prefix(mk-SynAct(lab, d, t), P), \rho), cit) \triangleq$$

$$\text{let } dv = VS[d](\rho) $$

$$\text{ delays } = \{(dt + dv + cit) | dt \in TIME\} \text{ in}$$

$$\{[]\} \cup \text{union } \{\{([lab, delay])\}^* t | t \in TR((expr, \rho'), 0) \cdot$$

$$\rho' = \mu(\rho, Cvenv \mapsto \rho.Cvenv \cup \{t \mapsto delay\})$$

$$\text{ delays } \}$$

First the set of delay values $delays$ is computed. Then for each delay value in the set a set of traces is is computed, that takes the delay value into account. The result is then computed by taking the union of all sets of traces. The rule applies also to the output prefixes.

An equivalent function can be defined that takes the effect of input prefixes into account.
\( Tr : STS \times TIME \rightarrow \text{TTRACE-set} \)

\[
Tr((\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, t, P), \rho), \text{cit})) \triangleq \\
\text{let } (-, \text{type}) = \text{PortMap}(\text{lab})(\rho), \\
\text{vals} = \{v \mid v \in \text{VAL} \cdot \text{showtag}(\text{type}) = \text{showtag}(v)\}, \\
\text{delays} = \{(dt + dv + \text{cit} \mid dt \in \text{TIME}\} \text{ in} \\
\{[]\} \cup \\
\text{union } \{\text{union } \{\text{[(lab, delay)]} \权利 t \mid t \in \text{TR}((e, \rho'), 0) \cdot \\
\rho' = \mu(\rho, C_{\text{env}} \rightarrow p.C_{\text{env}} \cup \{t \rightsquigarrow \text{delay}\})\} \\
\mid (e, \rho) \in \text{PRE}_{in}(\sigma, \text{val})\} \\
\mid \text{val} \in \text{vals} \land \text{delay} \in \text{delays}\} \\
\]

The function \( \text{PortMap} \) maintains a map between a port name and its associated capability and type.

The idling prefix is distributive over the choice operator, i.e. the following rule holds:

\[
\text{idle } (t) \odot (P \oplus Q) = (\text{idle } (t) \odot P) \oplus (\text{idle } (t) \odot P) \tag{6.1}
\]

Semantically idling is not an unique property of the idling prefix. The other prefixes and the choice and composition expressions can idle as well. Rule 6.1 reflects the fact that idling in choices is equivalent with idling the same amount of time in the operands of the choice. A direct result of this rule is the following property of \( TR \):

\[
\text{TR}((\text{mk-choice}(L, R), \rho), d) \triangleq \text{TR}((L, \rho), d) \cup \text{TR}((R, \rho), d) \tag{6.2}
\]

**Example 6.6** Consider the next agent expression.

\[
(\text{idle } (5) \odot p \odot \text{null}) \oplus (\text{idle } (3) \odot q \odot \text{null})
\]

The behaviour of this agent is determined by the progression of time. It is divided into three intervals, \( A \), \( B \) and \( C \).

\[
\begin{array}{ccc}
A & B & C \\
\circ & \circ & -
\end{array}
\]

\[
0 & 3 & 5
\]

In interval \( A \) the agent can only idle, in \( B \) the \( q \) action in enabled and in \( C \) both \( p \) and \( q \) actions are enabled. Its set of traces becomes:

\[
\{[]\} \cup \\
\{[(p, t)] \mid t \in \text{TIME} \cdot t \geq 5\} \cup \\
\{[(q, t)] \mid t \in \text{TIME} \cdot t \geq 3\}
\]

\( \square \)
6.3. Trace analysis

The traces of a combination of idling and composition do not follow from a simple distribution property. The maximal progress property of the time semantics dictates that whenever an internal action is possible, it will proceed without idling. As such idling distributes only over composition \((P \mid Q)\) on a time interval in which no internal action between \(P\) and \(Q\) are possible. So, interleaving traces alone is not appropriate. The resulting set of interleaved traces must be restricted to those actions that can occur before any internal action:

\[
TR((P \mid Q), \rho_p \parallel \rho_q), i) \triangleq \\
\text{let } rset = \{(a, t) \mid a \in \text{Sort}_i(P) \cap \text{Sort}_i(Q) \cdot t \geq l\} \text{ in} \\
TR((P, \rho_p), i) \bowtie TR((Q, \rho_q), i) \triangleq rset
\]

where \(\text{Sort}_i(E)\) defines the set of possible actions of behaviour expression \(E\) after idling \(l\) time units.

**Example 6.7** The choice expression from the last example in combination with a composition, e.g.

\[
((\text{idle } 5) \odot p \odot \text{null}) \oplus (\text{idle } 3) \odot q \odot \text{null}) \ | (p \odot \text{null})
\]

behaves as follows. First the interleaved traces are computed:

\[
\left(\begin{array}{c}
\{([]\} \cup \\
\{([p, t]) \mid t \in \text{TIME} \cdot t \geq 5\} \cup \\
\{([q, t]) \mid t \in \text{TIME} \cdot t \geq 3\},
\end{array}\right) \bowtie \left(\begin{array}{c}
\{([],\} \cup \\
\{([p, t]) \mid t \in \text{TIME}\}
\end{array}\right)
\]

which results in

\[
\{([],\} \cup \\
\{([p, t]) \mid t \in \text{TIME}\} \cup \\
\{([q, t]) \mid t \in \text{TIME} \cdot t \geq 3\} \cup \\
\{([p, t], (p, u)) \mid t, u \in \text{TIME} \cdot t < 5 \Rightarrow t + u \geq 5\} \cup \\
\{([p, t], (q, u)) \mid t, u \in \text{TIME} \cdot t < 3 \Rightarrow t + u \geq 3\} \cup \\
\{([q, t], (p, u)) \mid t, u \in \text{TIME} \cdot t \geq 3\}
\]

The restriction results in

\[
\{([p, t]) \mid t \in \text{TIME} \cdot t \geq 5\}
\]

and after the restriction the resulting set of traces becomes

\[
\{([],\} \cup \\
\{([p, t]) \mid t \in \text{TIME} \cdot t < 5\} \cup \\
\{([q, t]) \mid t \in \text{TIME} \cdot t \geq 3\} \cup \\
\{([p, t], (q, u)) \mid t, u \in \text{TIME} \cdot t < 5 \land u < 3 \Rightarrow t + u \geq 3\} \cup \\
\{([q, t], (p, u)) \mid t, u \in \text{TIME} \cdot t \geq 3\}
\]
Restriction and relabeling obey the simple laws

\[ TR((\text{mk-Restriction}(P, \text{ports}), \rho), i) \triangleq TR((P, \rho), i) \triangleq \{ (p, -) \mid p \in \text{ports} \} \]

and

\[ TR((\text{mk-Relabeling}(P, \text{rmap}), \rho), i) \triangleq TR((P, \rho), i) \uparrow \text{rmap} \]

The null agent delivers the empty trace such that

\[ TR((\text{mk-NullAgent}(-, -), \rho), i) \triangleq [\cdot]. \]

The second important factor in the computation of the traces of an agent is the inherent looseness in that agent. All environment manipulation result in internal transitions, that may not idle. The resulting effect of the loose transition rules is captured in the following equation:

\[ TR(\sigma, i) \triangleq \text{union} \{ \{ t \mid t \in TR(sts, i) \} \mid sts \in SE(\sigma) \} \]

where \( \sigma = (\text{bezpr}, \rho) \) such that \( \text{bezpr} \) is either a \textit{Spec} or a \textit{AgentService} or a \textit{UnitService} or a \textit{AgentValueLet} or a \textit{AgentLetBe} or a \textit{AgentAgentLet} or a prefix with a SM or a AgentIf construction.

### 6.3.4 Traces of agents with cyclic behaviour

The function \( TR \) fails to compute the trace of an agent with cyclic behaviour. Agent behaviour definitions like

\[ A \triangleq a \circ A \]

are normal definitions, which are certainly valid and exhibit an infinite behaviour. It is not a recursive behaviour definition in the sense of the recursion of CSP or CCS. In these latter two notations recursion forms a specific property of the notation which is given a meaning in an equational setting by substitution. In CSP the equation \( X = F(X) \) has an unique solution, denoted by \( \mu.X : F(X) \), where \( X \) denotes a process variable, \( A \) the alphabet of the process \( X \) and \( F \) a function resulting in a guarded behaviour expression. This unique process is defined using standard least fixed point techniques (see e.g. [103]). CCS has a similar concept. As MOSCA has no equational theory, the concept of recursion can not easily be incorporated. Hence, the intuition of cyclic behaviour in MOSCA is not based on recursion but on behaviour continuation addressed by agent service through an environment that contains bindings from agent identifiers to guarded behaviour expressions. Now the question rises by what means
can the traces of agents with cyclic behaviour be characterised? It is stated earlier that traces are finite and thus record the evolution of a *finite* part of the behaviour of cyclic agents. The function \( TR \) can be simply extended to capture the length of the traces computed thus far. Let \( TR_l \) denote the function that computes all traces up to length \( l \). Now all traces of an cyclic agent can be defined as follows:

\[
TR(\sigma, i) = \bigcup_{i \geq 0} TR_l(\sigma, i)
\]

where each \( TR_l \) is a function that computes its result in a finite number of steps. \( TR_l \) can be simply derived from the definition of \( TR \) in the last subsection by adapting the prefix handling cases. To show that two cyclic processes have the same traces one has to show that the two processes have the same partial traces, that is,

\[
TR(\sigma_1, 0) = TR(\sigma_2, 0) \iff \forall l \geq 0 \cdot TR_l(\sigma_1, 0) = TR_l(\sigma_2, 0).
\]

### 6.3.5 Analysis by traces

The various constraints defined over a certain software component modelled in MOSCA can be expressed as simple first order predicates over the set of (timed) traces of the specification executing in a predefined environment \( \rho \). The first subsection addresses the phenomena that timed traces of a software component capture all possible action sequences in time, irrespective of the capabilities of its environment.

The applicability of traces is limited in the context of termination and deadlock. This observation is clarified in the second subsection.

#### 6.3.5.1 Traces and constraints

Traces can be used to record proof obligations over agent specifications induced by both behavioural and timing constraints. Often these two kinds of constraints are strongly connected. To illustrate the capabilities of traces in this context a simple example is studied in some detail.

Figure 6.12 represents schematically a triple modular redundancy system (TMR). Its functionality and timing constraints are as follows. The TMR system contains a source process \( S \), a collector process \( C \) and three processes \( P_i \) that compute the same function. The sender sends at the same moment in time each of the three \( P_i \) processes the same value. Each process \( P_i \) computes a function \( r = f(s) \) and delivers the result \( r_i \) on its output port exactly 5 time units after the reception of the input value \( s_i \).
Whenever the collector receives two identical values on two of its three input ports at the same moment in time it copies this value to its output port o.

Although the TMR system is quite simple, it captures the essential qualities of time behaviour constraints. A MOSCA model is given in specification 6.6. The top level agent is TMR which offers the composition of the five processes shown in figure 6.12. Agent S passes the value v, after reception, in parallel at the same moment to each of its tree output ports. Agent P merely computes the value f(v), idles 5 time units and offers the result on its output port. Unit C receives the computed values, compares the values and the associated time of reception, and if appropriate copies the received value to its output port.

The external observable behaviour of S is captured in the following propo-
6.3. Trace analysis

Proposition 6.2 (Sender constraints)

let \( tr4 = \{ t \mid t \in TR((S, \rho), 0) \cdot \text{len } t = 4 \} \) \in \( \forall t \in tr4 \).

let \( sp = \{ p \mid p : \mathbb{N} \cdot ((p - 1) \mod 4 = 0) \land p < \text{len } t \} \) \in \( \land_{p \in sp} \text{check}_-\text{quad}_r(\text{extr}_-\text{quad}_r(p, t)) \)

For each trace that contains a multiple of four elements it must be so that each successive subtrace of four elements passes the checkes formulated in the function \( \text{check}_-\text{quad}_r \).

\[
\text{check}_-\text{quad}_r \ (qt : TTRACES) \ r : \mathbb{B}
\]

\pre \text{len } t = 4

\post let \( as = \{ a \mid ((a, t)) \in \text{elems } qt \} \) \in

let \( ts = \{ t \mid ((a, t)) \in \text{elems } qt \} \) \in

\( \text{card } as = 4 \land \text{card } ts = 1 \land \)

let \( ((fa, t)) = \text{hd } qt \) \in

\( fa = \text{in} \)

\( \square \)

\begin{align*}
S & \triangleq \\
\text{ports in } & \text{in } : \mathbb{R} \\
\text{out } & \overline{s1}, s2, s3 : \mathbb{R} \\
\end{align*}

\begin{align*}
\text{let } & S1 = s1(v) \odot r \odot \text{null in} \\
\text{let } & S2 = s2(v) \odot r \odot \text{null in} \\
\text{let } & S3 = s3(v) \odot r \odot \text{null in} \\
\text{let } & R = r \odot r \odot r \odot S \text{ in} \\
\text{in}(v) \odot ((S1 \mid S2 \mid S3) \mid R) \setminus \{r\}
\end{align*}

The comparison of the set of traces defined in constraint 6.2 with the traces of specification 6.7 reveals a mismatch between the constraint and the specification which is typical for MOSCA. The traces of Sender encompass not only the traces such as defined by constraint 6.2 but all traces built from quadruples defined by

\[
\{ qt \mid qt \in TTRACES \cdot \text{len } qt = 4 \land \\
\text{let } as = \{ a \mid ((a, t)) \in \text{elems } qt \} \in \\
\text{card } as = 4 \land \\
\text{let } ((fa, t)) = \text{hd } qt \text{ in} \\
fa = \text{in} \}
\]
i.e. without fulfilling the timing constraint. The behaviour of the agent $S$ is not only dependent of the internal structure of the agent $S$ but also on the behaviour of the environment. $S$ is capable to offer the three synchronizations to its environment, and under the assumption that the environment can accept all three synchronizations without any time delay it will behave correctly with respect to constraint 6.2.

The behaviour of each process $P$ is straightforward. $P\text{-}Failure$ models an agent that may pre-empt the normal behaviour of $P$ such that $P$ starts to diverge. As such, the component $P$ may fail to compute the value $f(v)$ and subsequently send it to the collector.

The next constraint specification and MOSCA implementation reveal another important issue of constraints defined by trace sets. It is often not easily checkable whether the trace set is restrictive enough. Suppose that the behaviour of the collector is characterised by the next proposition over its set of timed traces.

**Proposition 6.3** (Collector constraint)

$$
\forall t \in TR_k((C, \rho), 0) \cdot
\begin{array}{l}
\text{let } tps = \left\{ (r_i, t), (r_j, t) \right\} \mid i, j \in \{1, 2, 3\} \land i \neq j \right\} \text{ in } \\
\text{let } ps = \bigcup \left\{ \{ p \mid p = pos(tp, t) \land \text{len } t \geq p + 2 \} \mid tp \in tps \right\} \text{ in } \\
\forall p \in ps \cdot t(p + 2) = \langle o(v), t + 2 \rangle
\end{array}
$$

The set $tps$ captures all traces of length 2 that will trigger the output of the Collector. The set $ps$ contains all positions in a trace $t$ that mark the occurrence of one of the elements of $tps$ such that $t$ offers at least another element after the occurrence. Specification 6.9 offers a MOSCA model for the collector that, again under the assumption of correct behaviour of the environment, behaves within the limits defined in constraint 6.3. However, careful inspection of the MOSCA model reveals that the agent is intuitively speaking certainly not performing the expected behaviour. Whenever two synchronizations are received, on the same time and with the same value, the value is send out and the agent starts anew. But what will happen to the third synchronization arriving at the same time with the same value?
6.3. Trace analysis

\[ \text{unit } C \]
\[ \text{ports in } r_1, r_2, r_3 : \mathbb{R} \]
\[ \text{out } \mathbf{out} : \mathbb{R} \]
\[ \text{shares } C_1 (\mathbb{N} \times \mathbb{T} \times \mathbb{R}) , \]
\[ C_2 (\mathbb{N} \times \mathbb{T} \times \mathbb{R} \times \mathbb{N} \times \mathbb{T} \times \mathbb{R}) \]

\[ C \triangleq \]
\[ r_1(v), \ast t_{11} \circ C_1 ((1, t_{11}, v)) \oplus \]
\[ r_2(v), \ast t_{12} \circ C_1 ((2, t_{12}, v)) \oplus \]
\[ r_3(v), \ast t_{13} \circ C_1 ((3, t_{13}, v)) \]

\[ C_1 ((n, t, v)) \triangleq \]
\[ r_1(v_2), \ast t_{21} \circ C_2 ((n, t, v, 1, t_{21}, v_2)) \oplus \]
\[ r_2(v_2), \ast t_{22} \circ C_2 ((n, t, v, 2, t_{21}, v_2)) \oplus \]
\[ r_3(v_2), \ast t_{23} \circ C_2 ((n, t, v, 3, t_{21}, v_2)) \]

\[ C_2 (n_1, t_1, v_1, n_2, t_2, v_2) \triangleq \text{if } n_1 \neq n_2 \land t_1 = t_2 \land v_1 = v_2 \]
\[ \text{then } \text{idle} \circ \text{out}(v_1) \circ C \]
\[ \text{else } C \]

It will be handled in a next C phase and will cause the collector to become completely out of phase.

In this case the collector constraint is clearly not stated strongly enough. It offers a necessary condition but fails to address the behaviour that deviates from the condition.

6.3.5.2 Traces and termination properties

As observed before, trace semantics do not distinguish between termination, deadlock and divergence. To the observer the agent that is in either one of these three states is silent and remains so forever.

To look for deadlock in the presence of actions means finding traces of specific length. The set of traces \( TR_l(\sigma, d) \) for \( l \) sufficiently large computes all traces of length \( l \), but does not states whether there are traces of smaller length at all. The set computes all the behaviour patterns without deadlock until so far, which is not the demanded result.

To look for deadlock in the absence of actions could be done by computing the following:

for each action \( a \) of interest do
  for each \( l \) up to a specific limit \( max_l \) do
    compute \( traceset = TR_l(\sigma, d) \),
    delete all traces from \( traceset \) that contain action \( a \)
if \( \text{traceset} = \{ \} \) then a deadlock is encountered

For this process to be successful a safe value for \( \text{max}_i \) must be found first, e.g. the length of the largest cycle that can occur in the behaviour. Finding this value is not a trivial problem itself.

### 6.4 Conclusion

State spaces and traces are complementary in many aspects.

- A state element captures one particular potential continuation behaviour and environment development in the evolution of a given \( \text{sts} \) configuration. A trace of a given \( \text{sts} \) configuration captures the external observable actions of a specific evolution up to a certain moment in time.

- The state space of a given \( \text{sts} \) configuration represents all possible states in the evolution of the \( \text{sts} \). The set of traces of a given \( \text{sts} \) configuration contains all possible observable action sequences in time up to some moment in time.

- The state space reveals the effects of the communication and the computations affecting the internal state of the agents.

- The trace set abstracts away from the internal state component of the agents.

- State spaces offer a way to analyse termination properties of agents.

- Trace sets offer a means to record behavioural and timing requirements.

### 6.4.1 State spaces in analysis

State space analysis is a much used technique in many state based specification notations that generate finite state spaces. E.g. finite state machines, state charts, PAISLey and Petri-Nets.

In the context of infinite state space generating notations, such as CCS and TCCS state space analysis is limited to finite subspaces of the total state space ([156], [157]). In the case of TCCS a particular semantic approach has been proposed lately for the regular subset based only on prefix and choice operators that reduces the state explosion due to the incorporation of time ([104]) to finite proportions. The semantics is still SOS based
but for each TCCS entity a new transition rule transforms the absolute timing construction into a symbolic counterpart. As a result the state space is collapsed more or less in the same manner as in $SS$, but with the exception that a kind of symbolic time component has remained available. Holmer c.s. have argued that this approach is not applicable for the whole of the TCCS notation. This line of research seems promising, but much work still needs to be done.

The contractions defined in this section all cover the complete erasure of some aspect of the MOSCA sts under inspection: all value manipulations, all timing aspects, all internal actions. It is not too complicated to define partial contractions that erase the specific aspects only partial. This can be achieved by introducing regions of a certain kind of scope over the total state space in which the contraction is defined. The scope region may contain states scattered over the whole space or be constructed as a closed region. The definition of the scope region can take the form of a predicate over the state type of the total state space (possible scattered result) or as a predicate over the $\rightarrow$ relation (closed result). In many cases these partial contractions will reveal the wanted properties without imploding the space in such form that many interesting properties are lost.

Total inspection of state spaces remains troublesome. For both full contractions or partial contractions the problem of generation still remains. If the calculation of the contracted state space takes the same order of steps than the computation of the total state space nothing is gained with respect to the realizability of the state space. On the other hand the contracted state spaces offer a strong proof technique when the inspection is limited to particular finite subspaces.

For inspection with respect to termination, deadlock and reachability properties state space generation is a strong tool. In my opinion the approach taken in the Edinburgh TCCS tool set is not satisfying in the sense that whenever the state space to be generated is infinite the computation never stops. To advance the scope of applicability of such tools a more flexible approach seems necessary. My experiences with the implementation of an experimental state space generator for MOSCA capable of generating partial contracted state spaces have shown that given the flexibility of partial contraction definitions it is not so hard to define such contractions that (i) interesting properties of the state space remain visible, and (ii) the cardinality of the state space is reduced to manageable size. Still much work has to be done both on the theoretical and practical aspects of contractions to deliver a practical and above all useful tool for analysis of MOSCA specifications.
6.4.2 Traces in analysis

The range of different trace semantics in the context of MOSCA offers the same possibility for abstraction as developed for state spaces. Explicit generation of the timed traces remains an impossible task, due to the possible infinity of trace sets. Implicit characterization through set comprehension in combination with first order predicate logic offers more potential for analysis.

A point of concern in the application of traces is the sheer amount of different traces that can occur even in the context of simple specifications. E.g. HOARE has shown ([103]) that for a CSP specification that addresses a variant of the dining philosophers that is expectedly dead-lock free, the number of involved traces becomes exponential in the order of the number of states (in the specific case $2^{1800000}$). In the case of timed traces the number of traces is even larger! Other experiments have shown similar observations ([43], [59]). Inspection of complete trace sets remain as such impossible.

Traces may play an important rôle in the context of agent refinement. Traces form the basis for a specific definition of agent equivalence, and as such can be used to establish the proof of equivalence between two agents. In this setting the invisibility of the state is a blessing as it is abstracted away in the process of establishing equivalence based only on observable behaviour.

The explicit generation of trace sets is not very practical. It may be the case that tools with a more generative character may offer better ways to analyze the trace sets. The sets of traces can be considered as languages. In this particular view the agent itself acts both as defining device and recognizing device. An associated formal grammar derived from the syntactical structure of the agent can act as generative device.

Future experiments of combinations of traces with logic devices to express time dependent behaviour are needed to investigate the analysis power of traces in more detail.

6.4.3 Other approaches

There are other techniques to analyze the behaviour of agents. Two other important techniques are logic verification of properties and simulation or execution of specifications.
6.4. Conclusion

6.4.3.1 Model checking with logics

In this area much research is still needed. Both VDM-SL and CCS come equipped with a logic to analyse properties of the specifications. These logics are completely different and the nature of their combination is not evident. The time concept is very complex to capture in a logic. A specific modal logic has been found to describe the regular subset of TCCS expressions build only from timed prefix and choice (see HOLMER c.s. in [104]). It seems to me that the approach taken in the case of state space analysis could serve also on logic model checking, that is, to use different logic devices to verify certain aspects of MOSCA specifications. In this context the semantic orthogonality of MOSCA constructions play an important rôle.

6.4.3.2 Model checking by simulation/execution

Model checking by simulation offers alternative to e.g. analytic performance models. Simulation may give insights in the real-time performance aspects of a model whenever analytic performance models fail e.g. due to complexity or intractability of the model. Even when an approximate model of the system is studied, simulation could be a valuable tool in validating assumptions of the approximation of the model. As such, simulation could be used quite early during the development of the model, and provide insights that aid further understanding.

In this context the executability of the notation in which the models are stated becomes an important factor. The ease by which an executable model can be produced, has an important influence on the feasibility of the approach. If it is relatively easy to produce an executable model it is often likewise easy to transform the specification into an implementation. On the other hand, although simulation may provide insights on the model, little knowledge may be gathered on the real executing environment of the implementation.

The VDM-SL notation on which MOSCA is based, is partially executable. Several experimental simulators or rapid prototypers have been developed (see e.g. [178] for an overview of tool building efforts and [185] part 2: Reports and part3: Tools description). The CCS notation is not strongly supported by simulators or rapid prototypers. In [80] the authors describe a tool that supports simulation of CCS specifications based both on equational term rewriting and SOS transitions. A more formal approach is taken in [57]. The Edinburgh Concurrency Workbench supports both static analysis tools and simulation.

Currently I am working on the various aspects of a MOSCA simulator / rapid-prototyper. Early results can be found in [170] and [171], in which a
subset translator is described that generates Ada source code from MOSCA specifications. A fairly large part of the functional subset of VDM-SL is covered, in combination with the most basic constructions to specify and manipulate state. Further the simulator offers all agent definition constructions. The execution model is based on Ada tasking semantics and incorporates a fair execution model. The model that was taken to implement the CCS operators is limited to single processor systems. This model is now being refined into a more general model of a MOSCA runtime kernel, which is not based on the Ada tasking semantics. A first implementation is currently under development. In chapter 8 I present a model that was designed specially for distributed systems.
7.1 Introduction

In this chapter the notation is set to work in two different settings. In the first case demonstration a software module of a large software system is specified. In the second case demonstration a complete system consisting of both hardware and software components is specified. The emphasis in both cases is centered around the modelling aspects of the case and not on the technical aspects of the involved application area.

The first case (section 7.2) describes a small part of a communication package for inter network node communication. It describes a simple communication protocol over a communication device that offers a restricted form of half-duplex communication. The case highlights the capabilities of the value part of MOSCA in combination with the process description capabilities. There is no time dependent behaviour incorporated.

The second case describes a simple model for a home heater system, as an example of a process control system. The specification is complete, that is, all components in the system are defined by MOSCA agents. It features a behaviour that is to a large extent time dependent. In this case all constituents of MOSCA are applied. It highlights in particular the capabilities of the timing constructions, in combination with both value part and process description techniques.

7.2 A communication protocol

This section will discuss various aspects of the specification and implementation of a simple communication protocol. This particular case study highlights several aspects of the application of MOSCA within the process
of software production. The role of the different analysis tools that were discussed in chapter 6 are illustrated by studying a concrete specification which forms a refinement of a more abstract specification. In section 7.2.1 the context of the case is shortly introduced. Section 7.2.2 introduces the case subject: a simple buffering mechanism that realizes a protocol for the usage of a transport medium that connects two processes operating in different computing environments, e.g. in two stations of different but connected computer networks. The section presents a small set of requirements for the behaviour of the transmission medium. The next section offers an initial rather abstract specification of the two way buffer. Section 7.2.4 presents a more concrete model of a realization of the requirements, in which a specific protocol is presented for the utilisation of the connection. In section 7.2.5 the concrete model is studied and its properties compared with the abstract model. The case study is closed with a short section on concluding remarks.

7.2.1 Introduction

One of the key concepts of distributed processing and computer networking is the need of entities in different systems to communicate. Entities may be anything capable of sending or receiving information and a system is understood as a physically distinct object that contains one or more entities. What is communicated, how it is communicated and when it is communicated must conform to some mutually acceptable set of conventions between the involved entities. The set of conventions is referred to as a protocol. Important characteristics of protocols are e.g. whether the communication between entities is direct or indirect, whether the protocol is monolithic or structured, whether it is symmetric or asymmetric and whether it conforms to a standard or not. The elements of a protocol may be divided into different classes. STALLINGS [198] distinguishes three major classes of elements in a protocol:

- **Syntax**: includes things as data format, coding, signals etc.
- **Semantics**: includes control information for coordination and error handling.
- **Timing**: includes speed matching and sequencing.

Communication proceeds through protocols on different levels of abstraction. In an application to application communication between two stations within different but connected networks there are several communication protocols (see fig. 7.1 based on [198]). At the application level two appli-
7.2. A communication protocol

Figure 7.1: Communication Protocols

cations may seem to communicate with each other (e.g. the Kermit file transfer service) when in fact the actual communication proceeds over a series of different protocols on different layers of abstraction.

7.2.2 Two-Way Transmission Medium

The case describes a simple process-to-process protocol that offers a two-way point-to-point transmission medium. The protocol is a typical example of an OSI session layer protocol (layer 5, see the ISO OSI standard [2]). A session between two parties may be divided into three units aimed at: (i) session establishment, (ii) dialogue management, and (iii) recovery. Subsequently, session protocols provide a means for two application processes to establish and use a connection. It provides a dialogue type, in this case two-way-alternate and a means for session recovery.

The case description will concentrate only on the second aspect of a session protocol. The two other units have a too strong dependence on other parts of the communicating system. The presentation of the dialogue management will suffice to make our points without the need to model the other parts.
The following list of items presents a partial description of the capabilities of the 2WTM.

REQ-1 The session protocol abstracts away from the syntax of the data exchanged between the two processes (which is normally covered by the presentation layer of the protocol) and offers a simple message type.

REQ-2 The two-way point-to-point transmission medium (further denoted by 2WTM) will be actualized through four different services granted to two parties: party-1 sending, party-1 reading, party-2 sending and party-2 reading of messages:

REQ-3 It is assumed that the transmission medium features a half-duplex connection that offers the two-way-alternate dialogue type.

REQ-4 The transmission medium is able to send and receive unbound but finite messages and delivers the messages in the order of reception.

REQ-5 The transmission medium is assumed to be infallible. The assumption of this last quality is justified by the assumption of the existence of a complex transport protocol on a lower level (OSI-level 4) that guarantees a reliable service.

REQ-6 The transmission medium establishes a certain amount of fairness with respect to the usage from both parties. It will not be the case that the 2WTM is continuously serving one party while the other party is waiting for service.

REQ-7 The transmission medium guarantees a minimum service with respect to timely respons, and realizes a certain amount of through-put.

Each of these items will be covered incrementally within the evolving specification.

7.2.3 An abstract 2WTM model

7.2.3.1 Modeling REQ-1 and REQ-2

On the most abstract level the interface of the MOSCA agent resembling the two-way transmission medium could be realized as:

\[ Message = \text{Byte}^* \]
7.2. A communication protocol

2WTM

ports in send1 : Message
  out receive1 : Message
  in send2 : Message
  out receive2 : Message

7.2.3.2 Modeling REQ-3 and REQ-4

The message type is a sequence type, and subsequently finite. To model the ordered reception of messages a simple queue mechanism will suffice. Two queues are needed, for each sender one queue. Sending messages through send1 will result in queuing the messages to the rear of the message queue associated with the send1 source. Receiving (reading) through receive1 will result in the removal of the head of the queue associated with send1. Initially the two buffers will be empty.

2WTM

ports in send1 : Message
  out receive1 : Message
  in send2 : Message
  out receive2 : Message

shares 2WTMact (Message* × Message*)

2WTM ≡ 2WTMact ([], [])

Again the interface hides the internal buffers from the environment. The internal interface consists initially of a value part holding two empty buffers. The only possible actions here are sending:

2WTMact (([], [])) ≡
  send1(m) ⊕ 2WTMact (([m], [])) ⊕
  send2(m) ⊕ 2WTMact (([], [m]))

The choice operator models the half-duplex quality of the transmission medium. The offered service implements the two-way alternate dialogue type in a natural way. Each sending action results in the instantaneous deliverance of the message at the message-buffer of the reader whereas the actual reading (i.e. usage) of the message is decoupled from the reception moment. After completion of a sending action the other party may continue with sending without any special protocol needed.

1 A full-duplex medium could be easily modelled by changing the choice operator into a composition.
The other three cases of buffer states are recorded next: one empty buffer and one non-empty buffer for both sending originators and two non-empty buffers. The full specification is given in spec. 7.1.

7.2.3.3 Modeling REQ-5 and REQ-6

The infallibility of the transmission medium does not require any additions to the specification developed so far. Messages cannot be lost. Each offered message will eventually be delivered as the sequence of messages waiting to be send out is bound.
The fairness aspect is not easily incorporated on this level of specification. The MOSCA sos semantics is not fair with respect to semantics of the choice operator. The model for this requirement is postponed to the next section where the fairness aspect is integrated in the protocol for the utilisation of the buffer.

7.2.3.4 Modeling REQ-7

It is not acceptable to wait for a very long time on the deliverance of a message at a remote site after accepting it at a local site. To model this requirement more accuracy is needed in the formulation of the requirements REQ-7. To accommodate this fact the rules for constructing real-time timing constraints given by DASARATHY could be applied. There are no R-R or R-S requirements, as there is no causal relationship between two subsequent receive actions, or between a receive and send action. There can be given two equivalent S-S requirements:

S-S (1) the maximum time in between the acceptance of two subsequent send1 actions is \( \tau_1 \) seconds.

S-S (2) the maximum time in between the acceptance of two subsequent send2 actions is \( \tau_2 \) seconds.

The time spent by 2WTM on handling an action, either sending or receiving must be restricted to a reasonable value. Given that sending and receiving take maximum \( \tau_s \) and \( \tau_r \) seconds it is still not possible to express the maximum waiting time \( \tau_1 \) in terms of \( \tau_s \) and \( \tau_r \), due to the lacking of fair behaviour. It is clear that e.g.

\[
\tau_1 < k\tau_s + l\tau_r \quad (7.1)
\]

\[
\tau_2 < m\tau_s + n\tau_r \quad (7.2)
\]

There are no direct stimulus-response requirements, as there is no direct causal relation between e.g. a send1 action and a receive1 action. There is
### 7.2. A communication protocol

<table>
<thead>
<tr>
<th>$Message = Byte^*$</th>
<th>7.1 2W-Transmission Medium</th>
</tr>
</thead>
</table>

**2WTM**

- **Ports in** $send_1$ : $Message$
- **out** $receive_1$ : $Message$
- **in** $send_2$ : $Message$
- **out** $receive_2$ : $Message$

Shares $2WTM_{act} (Message^* \times Message^*)$

$2WTM \triangleq 2WTM_{act} ([], [])$

$2WTM_{act} (\langle\langle [], [] \rangle\rangle) \triangleq$

- $send_1(m) \odot 2WTM_{act} (\langle\langle [m], [] \rangle\rangle) \oplus$
- $send_2(m) \odot 2WTM_{act} (\langle\langle [], [m] \rangle\rangle)$

$2WTM_{act} (\langle\langle [z] \leadsto s, [] \rangle\rangle) \triangleq$

- $send_1(m) \odot 2WTM_{act} (\langle\langle [z] \leadsto s \leadsto [m], [] \rangle\rangle) \oplus$
- $send_2(m) \odot 2WTM_{act} (\langle\langle [z] \leadsto s, [m] \rangle\rangle) \oplus$
- $receive_1(z) \odot 2WTM_{act} (\langle\langle s, [] \rangle\rangle)$

$2WTM_{act} (\langle\langle [], [z] \leadsto s \rangle\rangle) \triangleq$

- $send_1(m) \odot 2WTM_{act} (\langle\langle [m], [z] \leadsto s \rangle\rangle) \oplus$
- $send_2(m) \odot 2WTM_{act} (\langle\langle [], [z] \leadsto s \leadsto [m] \rangle\rangle) \oplus$
- $receive_2(z) \odot 2WTM_{act} (\langle\langle [], s \rangle\rangle)$

$2WTM_{act} (\langle\langle [l] \leadsto ls, [rx] \leadsto rs \rangle\rangle) \triangleq$

- $send_1(m) \odot 2WTM_{act} (\langle\langle [l] \leadsto ls \leadsto [m], [rx] \leadsto rs \rangle\rangle) \oplus$
- $send_2(m) \odot 2WTM_{act} (\langle\langle [l] \leadsto ls, [rx] \leadsto rs \leadsto [m] \rangle\rangle) \oplus$
- $receive_2(rx) \odot 2WTM_{act} (\langle\langle [l] \leadsto ls, rs \rangle\rangle) \oplus$
- $receive_1(lz) \odot 2WTM_{act} (\langle\langle ls, [rx] \leadsto rs \rangle\rangle)$
a temporal constraint that says that each send1 action should eventually be matched by a receive1 action in order to prevent forever growing message buffers and lost messages.

### 7.2.4 A concrete model of the 2TWB

In this section a refinement of the former realization of the 2WTM agent is presented.

1. A first refinement concerns the unlimited (although finite) length of both messages and mailboxes:
   - (a) message lengths should be bound to manageable pieces,
   - (b) mailbox capacity for both sender1 and sender2 messages should be bound.

   The messages are normally divided into pieces of fixed length and encapsulated by a session protocol data unit (SPDU) which is being send over through the transport link.

2. The two-way alternate mode is elaborated upon. A realization of the half-duplex quality of the transmission link will result in further constraints on the model.

3. The fairness constraint is realized.

   In figure 7.2 a simple connection scheme is sketched that applies a connection medium built from two two-way-buffers (TWB's) and a sender/reader combination that forms the interface between the clients of the 2WTM and the buffers. The TWB's perform two tasks. The connected buffers act as a switch that creates a directed channel between a sender/reader combination. The dotted arrows denote such a combination. The switch should prohibit sender/sender and reader/reader connections as depicted by dashed lines. Also a local combination of sender and reader is prohibited from any communication. Further the buffers act as concentrators by gathering data from senders element by element, packing the elements in blocks of fixed size, encapsulating the data in a SPDU and sending the SPDU along. At the receiving side the data is unpacked and restored into the original message. Sending and reading take place after a protocol setup through synchronization of the involved agents. The discussion will concentrate first on the realization of the transport of a message through a connected sender/reader combination. Secondly the actions are proposed to realize the mechanics of the switch from one sender/reader to the other sender/reader combination.
7.2. A communication protocol

![Diagram showing Sender/Receiver Connection Scheme]

Figure 7.2: A simplified Sender/Receiver Connection Scheme

7.2.4.1 A protocol for sending messages

There are two different action series possible, (i) first a sender becomes ready to send a series of values, or (ii) a reader becomes ready first to accept a series of values. In the context of these protocols the action series will further be denoted by the term *scenario*. Below a synchronization protocol for both scenarios is sketched. The system side on which the first action is taken, either by a sender or reader, is said to contain the *local two-way-buffer* thus making the other buffer known as the *remote two-way-buffer*. The scenario is recorded in a simple tabulated notation consisting of four parts:

- a set of scenario steps,
- *SE*, the step expression, a specification of the possible sequences of scenario steps that may occur,
- *IR*, an invariant relation that constrains the possible values of the repeaters in the step expression and
- *SC*, the successor condition, that controls the closures in the step expression.

Let's examine the Sender-First scenario in some detail. Initially, a two-way-buffer is ready to accept either a sender or a reader request for action, from local or remote agents, giving 2 different behaviour patterns, as the
<table>
<thead>
<tr>
<th>Lab</th>
<th>LS</th>
<th>LB</th>
<th>RB</th>
<th>RR</th>
<th>Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>lsr → lsr</td>
<td></td>
<td></td>
<td></td>
<td>local sender ready</td>
</tr>
<tr>
<td>s2</td>
<td>rsr → rsr</td>
<td></td>
<td></td>
<td></td>
<td>signalling the remote TWB that a remote sender is ready to send</td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td>lrr ← lrr</td>
<td></td>
<td></td>
<td>local reader ready</td>
</tr>
<tr>
<td>s4</td>
<td></td>
<td>rrr ← rrr</td>
<td></td>
<td></td>
<td>signalling the local TWB that a remote reader is ready, the connection is established</td>
</tr>
<tr>
<td>s5</td>
<td>put → get</td>
<td></td>
<td></td>
<td></td>
<td>sender passes values to buffer</td>
</tr>
<tr>
<td>s6</td>
<td></td>
<td>snd → rec</td>
<td></td>
<td></td>
<td>after the buffer gets full, the L-TWB sends it to the R-TWB</td>
</tr>
<tr>
<td>s7</td>
<td>put → get</td>
<td></td>
<td></td>
<td></td>
<td>the local TWB accepts values,</td>
</tr>
<tr>
<td>s8</td>
<td></td>
<td>put → get</td>
<td></td>
<td></td>
<td>the remote TWB passes values to the reader</td>
</tr>
<tr>
<td>s9</td>
<td>eos → eos</td>
<td></td>
<td></td>
<td></td>
<td>sender signals end,</td>
</tr>
<tr>
<td>s10</td>
<td>put → get</td>
<td></td>
<td></td>
<td></td>
<td>the remote TWB passes values to the reader</td>
</tr>
<tr>
<td>s11</td>
<td></td>
<td>snd → rec</td>
<td></td>
<td></td>
<td>L-TWB sends last block</td>
</tr>
<tr>
<td>s12</td>
<td>put → get</td>
<td></td>
<td></td>
<td></td>
<td>the remote TWB passes last values to the reader</td>
</tr>
<tr>
<td>s13</td>
<td></td>
<td>eor → eor</td>
<td></td>
<td></td>
<td>the buffer signals end of message</td>
</tr>
</tbody>
</table>

\[
SE_{SF} = s_1 . s_2 . s_3 . s_4 . s_5^{*_{sn}} . \\
( s_6 . (s_7^{*_{sk}} || s_8^{*_{sl}}) )^{*_{sp}} . \\
( s_9 || (s_10)^{*_{sm}} ) . s_{11} . (s_{12^{*_{so}}}) . s_{13}
\]

\[
IR_{SF} = \begin{cases} 
    sp = 0 \Rightarrow sn < maxbuf \land sm = 0 \land so = sn \\
    sp > 0 \Rightarrow sl + sm = maxbuf \land so = sk 
\end{cases}
\]

\[
SC_{SF} = \begin{cases} 
    s_5 \rightarrow s_6 & \text{when } sn = maxbuf \\
    s_8 \rightarrow s_6 & \text{when } sk = maxbuf 
\end{cases}
\]

Figure 7.3: Sender-First Scenario
behaviours for local and remote actions are symmetrical. The switch component of the two-way-buffer restricts the actual connection to one of the sender/reader pairs. The first 4 steps resemble the initial phase of the protocol. Here the local sender signals the TWB of its wish to send a message through. Steps s2 to s4 establish the handshake with the remote TWB. Each put–get synchronization in step s5 transports a byte from the sender to the TWB. The invariant relation states, o.a. that if sn from the synchronization in step s5 is less than the maximum number of elements that the buffer can hold, it should be directly followed at the local side of the TWB combination by the eos–eos synchronization. The synchronizations in step 7 and 8, sk + sl in number, may appear in any order, denoted by the ‘||’ operator. The protocol proceeds to step s9 when after the skth put–get synchronization, such that sk < mazbuf the synchronization eos–eos appears. Whenever sk reaches the value mazbuf in step 7, the protocol will proceed with step 6. When in step 9, either the remote buffer is ready to accept the last block, when sl = mazbuf ∧ sm = 0, or there are still bytes to be passed to the reader, then sm > 0, but the total of sl + sm is the contents of the buffer, i.e. mazbuf. After sending the last buffer in step 11, containing the last sl bytes passed to the buffer in the last recurrence of step 7, the remote TWB passes these bytes to the reader component.

The scenario in which the reader comes first (fig. 7.4) is to a large extend symmetrical to the sender first scenario. It lacks the confirmation of a remote sender to the local side of the system.

7.2.4.2 The specification of the sender and reader agent

The scenarios form an elegant starting point for the design of a MOSCA realization of the protocol. From the protocol steps and the step expressions we can deduce action sequences for the actions within a single agent. E.g. the sender agent action sequences can be characterised by:

\[ \text{sender} = 1sr . \text{put}^* . \text{eos} \]

(7.3)

We can deduce the action sequences of the reader agent in a similar way.

\[ \text{reader} = 1rr . \text{get}^* . \text{eor} \]

(7.4)

The scenarios did not specify the service to the clients of the send and receive ports. The service for the send port could be modelled directly as:

\[ \text{Sender} \triangleq \text{send}(m) \odot \text{SenderA} ([], m) \]
<table>
<thead>
<tr>
<th>lab</th>
<th>LR</th>
<th>LB</th>
<th>RB</th>
<th>RS</th>
<th>Synchronization annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>lrr</td>
<td>lrr</td>
<td></td>
<td></td>
<td>local reader ready</td>
</tr>
<tr>
<td>r2</td>
<td>rrr</td>
<td>rrr</td>
<td></td>
<td></td>
<td>signalling the local TWB that a remote reader is ready</td>
</tr>
<tr>
<td>r3</td>
<td>lsr</td>
<td>lsr</td>
<td></td>
<td></td>
<td>local sender ready, the connection is established</td>
</tr>
<tr>
<td>r4</td>
<td>get</td>
<td>put</td>
<td></td>
<td></td>
<td>sender passes values to buffer</td>
</tr>
<tr>
<td>r5</td>
<td>rec</td>
<td>snd</td>
<td></td>
<td></td>
<td>after the buffer gets full, the R-TWB sends it to the L-TWB</td>
</tr>
<tr>
<td>r6</td>
<td>get</td>
<td>put</td>
<td></td>
<td></td>
<td>the local reader accepts values, the remote sender sends values</td>
</tr>
<tr>
<td>r7</td>
<td>get</td>
<td>put</td>
<td></td>
<td></td>
<td>sender signals end,</td>
</tr>
<tr>
<td>r8</td>
<td>eos</td>
<td>eos</td>
<td></td>
<td></td>
<td>the local reader accepts values</td>
</tr>
<tr>
<td>r9</td>
<td>get</td>
<td>put</td>
<td></td>
<td></td>
<td>R-TWB sends last block</td>
</tr>
<tr>
<td>r10</td>
<td>rec</td>
<td>snd</td>
<td></td>
<td></td>
<td>the local TWB passes last values to the reader</td>
</tr>
<tr>
<td>r12</td>
<td>eor</td>
<td>eor</td>
<td></td>
<td></td>
<td>the buffer signals end of message</td>
</tr>
</tbody>
</table>

\[
SE_{SF} = s_1.s_2.s_3.s_4^{\text{rm}}. \\
s_5.(s_6^{\text{sr}}||s_7^{\text{srk}}))^{rp}. \\
s_8.s_9^{\text{rm}}.s_{10}.s_{11}^{\text{ro}}.s_{12}
\]

\[
IR_{SF} = \begin{cases} 
\text{rp} = 0 & \Rightarrow \text{rn} < \text{maxbuf} \land \text{rm} = 0 \land \text{ro} = \text{rn} \\
\text{rp} > 0 & \Rightarrow \text{rl} + \text{rm} = \text{maxbuf} \land \text{ro} = \text{rk}
\end{cases}
\]

\[
SC_{SF} = \begin{cases} 
\text{s4} \rightarrow \text{s5} & \text{when } \text{rn} = \text{maxbuf} \\
\text{s7} \rightarrow \text{s5} & \text{when } \text{rk} = \text{maxbuf}
\end{cases}
\]

Figure 7.4: Reader-First Scenario
7.2. A communication protocol

\[ \text{SenderA} ((ms, m)) \triangleq \]
\[ \quad \text{if \ len} \ ms < \text{MaxMailbox} \]
\[ \quad \text{then send}(nm) \odot \text{SenderA} \ ((nm) \bowtie ms, m) \oplus \]
\[ \quad \text{SenderB} ((ms, m)) \]
\[ \quad \text{else SenderB} ((ms, m)) \]

where \text{MaxMailbox} models the capacity of the sender mailbox. The action sequence for the sender is mapped on:

\[ \text{SenderB} ((ms, m)) \triangleq \]
\[ \quad 1sr \odot \text{Sending} ((ms, m)) \]
\[ \text{Sending} (([], [])) \triangleq \text{eos} \odot \text{Sender} \]
\[ \text{Sending} ((ms \bowtie m, [])) \triangleq \text{eos} \odot \text{SenderA} ((ms, m)) \]
\[ \text{Sending} ((ms, [x] \bowtie s)) \triangleq \text{put}(x) \odot \text{Sending} ((ms, s)) \]

A similar realization for the reader agent is given below.

7.2.4.3 The specification of the two-way-buffer

Given the two protocols we can deduce action sequences for the two-way-buffer.

\[ \text{twb} = \text{lb-sf} \mid \text{rb-rf} \mid \text{rb-lb} \mid \text{lb-rf} \]  
\[ \text{lb-sf} = 1sr . rsr . rrr . \text{get}^+ . (\text{snd} . \text{get}^+)^* . \text{eos} . \text{snd} \]  
\[ \text{rb-rf} = rrr . 1sr . \text{get}^+ . (\text{snd} . \text{get}^+)^* . \text{eos} . \text{snd} \]  
\[ \text{rb-sf} = rsr . lrr . rrr . (\text{rec} . \text{put}^+)^* . \text{rec} . \text{put}^+ . \text{eor} \]  
\[ \text{lb-rf} = lrr . rrr . (\text{rec} . \text{put}^+)^* . \text{rec} . \text{put}^+ . \text{eor} \]

Combining the local-buffer sender-first action sequence \text{lb-sf} with the sequence for remote-buffer reader-first, \text{rb-rf} giving the action sequence for buffer-sending and similarly the remote-buffer sender-first action sequence \text{rb-sf} with the sequence for local-buffer reader-first, \text{lb-rf} giving the action sequence buffer-reading \text{b-r}, we get

\[ \text{twb} = \text{b-s} \mid \text{b-r} \]  
\[ \text{b-s} = (1sr . rsr . rrr) \mid (rxx . 1sr) \]  
\[ \quad . \text{get}^+ . (\text{snd} . \text{get}^+)^* . \text{eos} . \text{snd} \]  
\[ \text{b-r} = [rsr] . lrr . rrr . (\text{rec} . \text{put}^+)^* . \text{rec} . \text{put}^+ . \text{eor} \]

The specification for the two-way-buffer is built using status information describing the phases that occur during the two combined action sequences.
The states are depicted in figure 7.5. Without any sender or reader activity the two-way-buffer is idle. On arrival of a sender's request, given the fact that the switch position enables it sending, it becomes, after the appropriate synchronization protocol in a collecting/sending state at the sender's side whereupon the reader's side has entered a reader/delivering state. After the last data is sent, the final block has to be passed to the other two-way-buffer bringing the buffer back in idle state.

The realization of the switch function Each of the two TWB’s is capable of either receiving bytes from a sender and sending SPDU’s to the other TWB or receiving SPDU’s from the other TWB and after unpacking sending bytes to a reader. These two different behaviours are realized by a switch function. This switch is implemented by a piece of state information that records the current setting of the TWB: sending or receiving. The realization of the switching action is as follows.

- The state information is recorded in a state part of a MOSCA unit that encapsulates the three agents on each side of the TWB combination: a sender, a reader and a TWB. As a result a sender can be either active or inhibited, depending on the position of the switch.
- The TWB switch is given a specific position on initialization.
7.2. A communication protocol

- The sender with the inhibited status reacts on an incoming service request for sending with a request to the TWB combination to switch the sending/reading position.

- The implementation will grant the request immediately after the finalization of the next time a message is sent through.

- The TWB accepts requests to switch in the idle state. It registers the request to send (a sendreq action) in a local state part. After finalizing the next sending/reading action it switches the position. Hereeto signals the reading TWB arriving at the delivering last block state the sending TWB its request to switch. The sending TWB will be finished with the sending last block state and upon request will switch from sending to reading. If no such request is pending a simple continuation synchronization takes place.

7.2.4.4 The complete 2WTM specification

The complete realization of REQ-3 and REQ-6 is modelled below. Its structure reflects clearly the compositional style of MOSCA specifications (fig. 7.6). It is assumed that the two functions pack-SPDU and unpack-SPDU are defined in the environment. The specification of the agent 2WTM

![Figure 7.6: Structure of the 2WTM specification](image)

features the composition of two UNIT agents, one initialized in the sending mode and the other initialized in the receiving mode. The UNIT is composed of three agents, a sender, a reader and a two-way-buffer.

\[
\text{Mailbox} = \text{Message}^* \\
\text{inv-Mailbox}(s) \triangleq \text{len}< \text{MaxMailbox}
\]
Message = Byte

Byte = ...

StatusType = S | R

\[ \text{UNIT (StatusType)} \]

ports in \text{send} : Message

in \text{send2} : Message

out \text{receive1} : Message

out \text{receive2} : Message

\[ \text{UNIT (S)} \{ \text{send} \mapsto \text{send1}, \text{receive} \mapsto \text{receive2}, \text{snd} \mapsto \text{mchan}, \text{rec} \mapsto \text{mchan} \} \]

\[ \text{UNIT (R)} \{ \text{send} \mapsto \text{send2}, \text{receive} \mapsto \text{receive1}, \text{snd} \mapsto \text{mchan}, \text{rec} \mapsto \text{mchan} \} \]

in \{ \text{Unit1} | \text{Unit2} \}\{ \text{rsr, rrr, mchan, cont, psr} \}\{ \text{mm} \mapsto \text{snd}, \text{mm} \mapsto \text{rec} \}

unit \text{UNIT (StatusType)}

ports in \text{send} : Message

out \text{receive} : Message

\text{syn rsr, rrr, cont, psr}

out \text{snd} : Byte

in \text{rec} : Byte

state status : StatusType

\[ \text{UNIT (stat)} \]

let \text{Sender'} = \text{Sender}[\text{put} \mapsto \text{bchan}, \text{get} \mapsto \text{bchan}]

\text{Reader'} = \text{Sender}[\text{put} \mapsto \text{bchan}, \text{get} \mapsto \text{bchan}]

in ((\text{Sender'} | \text{Reader'} | \text{TWB (STAT)})

\{ \text{sreq, lsr, eos, put, lrr, eor, get} \})

where

SetStatus(stat: StatusType)

ext wr status : StatusType

post status = stat
7.2. A communication protocol

Sender
ports syn lsr
  syn sreq
  syn eos
  out put : Byte
  in send : Message
shares SenderA (Message* × Message),
  Sending (Message* × Message)

Sender ⊳ send(m) ⊙ SenderA ([] , m)

SenderA (⟨ms , m⟩) ⊳
  if len ms < MaxMailbox
    then send(nm) ⊙ SenderA (⟨[nm] ∼ ms , m⟩) ⊕ SenderB (⟨ms , m⟩)
    else SenderB (⟨ms , m⟩)

SenderB (⟨ms , m⟩) ⊳
  if status = S
    then lsr ⊙ Sending (⟨ms , m⟩)
    else sreq ⊙ lsr ⊙ Sending (⟨ms , m⟩)

Sending (⟨[ ] , [ ]⟩) ⊳ eos ⊙ Sender

Sending (⟨ms ∼ m , [ ]⟩) ⊳ eos ⊙ SenderA (⟨ms , m⟩)

Sending (⟨ms , [x] ∼ s⟩) ⊳ put(x) ⊙ Sending (⟨ms , s⟩)

Reader
ports syn lrx
  syn eor
  in get : Byte
  out receive : Message
shares Reading (Message* × Message),
  ReaderA (Message* × Message)

Reader ⊳ lrx ⊙ Reading (⟨[ ] , [ ]⟩)

Reading (⟨ms , m⟩) ⊳
  (eor ⊙ ReaderA (⟨[m] ∼ ms , [ ]⟩)) ⊕
  (get(c) ⊙ Reading (⟨ms , m ∼ [c]⟩))
\[ \text{ReaderA} \left( (ms \sim [m],[]) \right) \triangleq \]
\[ \text{if } \text{len } ms + 1 < \text{MaxMailbox} \]
\[ \text{then } \text{receive}(m) \circ \text{ReaderA} \left( (ms,[]) \right) \oplus \]
\[ \text{lrr} \circ \text{Reading} \left( (ms,[]) \right) \]
\[ \text{else } \text{receive}(m) \circ \text{ReaderA} \left( (ms,[]) \right) \]

\[ \begin{align*}
\text{unit } & TWB \left( \text{StatusType} \right) \\
\text{ports } & \text{syn } \text{sreq, lsr} \\
& \text{syn } \text{rsr, rrr} \\
& \text{syn } \text{eos, eor} \\
& \text{syn } \text{cont, psl} \\
\text{in } & \text{get} \quad : \text{Byte} \\
\text{out } & \text{put} \quad : \text{Byte} \\
\text{in } & \text{snd} \quad : \text{Byte}^* \\
\text{out } & \text{rec} \quad : \text{Byte}^* \\
\text{state } & \text{pendingSR} : \mathbb{B} \\
\text{shares } & \text{Idle} \\
& \text{ColSnd } (\text{Byte}^*) , \\
& \text{EXIT_CS} \\
& \text{RecDel}, \\
& \text{DNB } (\text{Byte}^* \times \mathbb{B}) \\
& \text{EXIT_RD} \\
\text{TWB } (\text{stat}) & \triangleq \sigma(\text{SetStatus(stat)}) \circ \text{Idle} \\
\text{Idle} & \triangleq \]
\[ \begin{align*}
\text{if } & \text{status } = S \\
\text{then } & (\text{lsl} \circ \text{rsl} \circ \text{rrr} \circ \text{ColSnd } ([]) ) \oplus \\
& (\text{rrr} \circ \text{lsl} \circ \text{ColSnd } ([]) ) \\
\text{else } & (\text{rsl} \circ \text{lrr} \circ \text{rrr} \circ \text{RecDel}) \oplus \\
& (\text{lrr} \circ \text{rrr} \circ \text{RecDel})) \oplus \\
& \text{srec } \circ \sigma(\text{pendingSR} : = \text{true}) \circ \text{Idle} \\
\text{ColSnd } (s) & \triangleq (\text{get}(c) \circ \text{let } \text{nbuf } = s \sim [c] \text{ in} \\
& \begin{align*}
\text{if } & \text{len } \text{nbuf } = \text{MaxLength} \\
\text{then } & \text{snd}(\text{pack-SPDU}(\text{nbuf})) \circ \text{ColSnd } ([]) \\
\text{else } & \text{ColSnd } (\text{nbuf}) ) \oplus \\
& \text{eos } \circ \text{snd}(s) \circ \text{EXIT_CS} \\
\end{align*} \]
\]
EXIT-CS ≜
(cont ⊗ Idle) ⊕
(psr ⊗ σ(status := R) ⊗ Idle

RecDel ≜ rec(s) ⊗ if len(buf) < MaxLength
then DNB ((unpack-SPDU(s), true))
else DNB ((unSPDU(s), false))

DNB ((s, last_block)) ≜ if s = []
then if last_block
then eor ⊗ EXIT-RD
else RecDel
else put(hs) ⊗ DNB ((tl s, last_block))

EXIT-RD ≜
if pendingSR
then psr ⊗ σ(SetStatus(SR); pendingSR := false) ⊗ Idle
else cont ⊗ Idle

where
values
MaxLength : N = 64

end
end TWB
end UNIT

7.2.5 Analysis of 2WTM

There are different strategies that can be of help during the process of establishing safety and liveness properties of the 2WTM models. E.g.:

- studying the effect of the presence of (certain) external events,
- studying the effect of the absence of (certain) external events and
- studying the effect of the values passed to/from (certain) external events.

After formulation the safety and liveness properties can be inspected through the application of various tools like trace analysis, logic analysis and state space analysis. The major subject of this section forms the analysis to investigate the adequacy of the refined 2WTM specification.
7.2.5.1 Safety and Lifeness properties of 2WTM

An important safety aspect of communication protocols is the absence of deadlock. This phenomenon can be present in various forms, e.g. as a situation in which no action is ever possible, or by a series of cyclic actions that together do not realize any progress at all in the operation of the protocol. Both manifestations ask for specific analysis strategies.

Trace analysis — In this case trace analysis is difficult to apply as both agents have infinite behaviour and the number of traces grows exponentially with the their lengths.

State space analysis — In the context of state spaces deadlock is associated with states from which no further actions are possible (final states) or with cycles in the graph without any progress. In the next two sections these two strategies are applied during the analysis of 2WTM. The complexity of finding final states depends solely on the size of the state space which is far less than the total number of traces.

In this case the size of the state space is manageable and I have chosen to apply this method in the analysis of 2WTM.

Besides deadlock in the general sense there are many other safety and lifeness criteria that can be derived from the concrete 2WTM specification. The unfulfillment of these criteria leads in general to a deadlocked state, but they provide additional analysis information. Many of these criteria follow from the fact that we deal with infinite cyclic behaviour within the defined agents.

Definition 7.1 General lifeness criterion for cyclic behaviour Let A be any agent behaviour expression embedded in a circular behaviour expression and let $First(A)$ be the set of actions that A can perform initially. Let $\alpha$ be any action of the surrounding agent expression such that it is restricted by a surrounding restriction. Then for all behaviour expressions in the form

$$\alpha \odot A$$

after the action $\alpha$ is taken eventually $a_i$ follows such that $a_i \in First(A)$. $\square$

As an example, $First(Sending) = \{put, eos\}$ and the expression

$$put(x) \odot Sending ((ms, s))$$

is surrounded by a restriction containing both put and eos thus a lifeness criterion for the Sending agent is:
7.2. A communication protocol

after taking a \texttt{put} action it must eventually be followed by either another \texttt{put} or an \texttt{eos} action.

Another kind of lifeness criterion can be derived from the \textit{SenderA} agent. Here an \textit{external action} is involved:

\[ \text{send}(\text{nm}) \odot \text{SenderA} (\ldots) . \]

Whenever the environment offers a \texttt{send} action eventually one of the actions from \textit{First}(\textit{SenderA}) must occur.

Safety criteria are related to both the behavioural part and the value manipulating part of \textsc{mosca}. In general it may not occur that any agent, function or operation computes \( \perp \) or more general that any semantic entity equals \( \perp \).

Each particular \textsc{mosca} specification raises besides these semantic safety criteria more specific safety criteria. E.g. in the case of the concrete \( 2 \textit{WTM} \) specification it must not be the case that the absence of external stimuli interrupts the possible flow of work. Another example of a specific safety criterion is the fact that is must never occur that the two buffers are in the same execution mode, both S or both R.

7.2.5.2 The state spaces of the abstract \( 2 \textit{WTM} \) model

First the total state space \( \textit{TSS} \) of the abstract \( 2 \textit{WTM} \) is analysed. The state space depends on the number of messages in the two buffers and the four capabilities of \( 2 \textit{WTM} \). The combined effect of the \textsc{mosca} value parts and the behavioural alternatives results in a state space of \( 4(m+1)(n+1) \) elements where \( m \) is the maximum number of messages in the \texttt{send1} buffer and \( n \) is the maximum number of messages in the \texttt{send2} buffer \footnote{The effect on the state space due to the contents of the messages itself is not taken in consideration here.}. There is no state divergence due to looseness. This total number of states is further constrained by the causal relation between the value parts and the behavioural alternatives to \( mn + 2(m+n+2) \) states. In figure 7.7 depicts symbolically the state space as result of all possible transitions:

- the state marked \((0,0)\) corresponds with the \( 2 \textit{WTM} \) in its initial execution state, \( 2 \textit{WTM}_{\text{act}} (\emptyset, [\emptyset]) \),

- all states on the rising diagonal starting from the initial state correspond to \( 2 \textit{WTM}_{\text{act}} (([x]^{\neg} s, [\emptyset])) \) for a specific length of the buffer in the first value of the value part,
Figure 7.7: Total \( mn \) state space abstract model
7.2. A communication protocol

- all states on the descending diagonal starting from the initial state correspond to \(2WTM_{act} \langle ([[],[x]\sim s]) \rangle\) and
- all other states correspond to \(2WTM_{act} \langle ([ls]\sim ls,[rx]\sim rs) \rangle\).

The reduced state space \(RCSS\) collapses into a four state derivation graph, depicted in figure 7.8. Each state corresponds to one of the four execution

![Diagram](image)

Figure 7.8: Reduced total state space abstract model

states in the \(2WTM\) specification.

The observational state spaces of the abstract \(2WTM\) coincide with the total state spaces of \(2WTM\) as there are no internal actions possible. Both state spaces contain no final states. After all, the behaviour of \(2WTM\) is cyclic and once initialized it never stops. This fact must be valid also for the state spaces of the refined model.

The state spaces of the refined \(2WTM\) model

It is obvious that the total state space of the refined \(2WTM\) model exceeds the number of states of the abstract model substantially. Let \(mmb\) be the maximum length of the message buffer in the sender and reader agents. Then the sender state space has a total of \(8mmb\) possible states resulting from 4 different execution states, a state predicate with two values and the mailbox with a capacity of holding \(mmb\) messages. The reader has a similar number of states in its state space, \(3mmb\) resulting from the three different execution states and the maximum capacity of the reader mailbox. The \(TWB\) can reach 16 different execution states and is depending on two state predicates, giving a total of 64 different states. The possible full total number of states for the \(2WTM\) equals

\[8^2 \cdot 3^2 \cdot 16^2 \cdot (mmb + 1)^2\]
resulting in $147456(m^2 + 1)^2$ states. Even with a restricted mailbox capacity of 1 message the number of states is substantial. However, the number is diminished significantly by the causal constraints dictated by the behaviour specifications of the sender, reader and TWB agents. The restricted total state space computes to 366 states containing 1411 transitions. This fact was established by constructing the spanning tree of the total transition graph. Although graphs of this size are still hard to comprehend visually it is not difficult to study the spanning tree and the action sequences that lead to the leaves of the tree. Doing so it appears that the tree contains a final node (nr. 59), i.e. a node with no outgoing transitions, corresponding to a dead-locked 2WTM state. It is the case that the refined specification is not correct after all! The action sequence on the path of the final state is the following:

$$\text{send1, } \tau, \text{lslr-lsr, } \tau, \text{send2, sreq-sreq, } \tau, \text{lrr-lrr},$$

which brings the 2WTM in the following state:

<table>
<thead>
<tr>
<th>Entity</th>
<th>MOSCA expression</th>
<th>state value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit1.status:</td>
<td></td>
<td>S</td>
</tr>
<tr>
<td>Sender1:</td>
<td>Sending $([], m)$</td>
<td></td>
</tr>
<tr>
<td>Reader1:</td>
<td>Reader</td>
<td></td>
</tr>
<tr>
<td>TWB1:</td>
<td>$\text{rrr} \odot \text{xxx} \odot \text{ColSnd} ([[]])$</td>
<td>false</td>
</tr>
<tr>
<td>TWB1.pendingSR:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TWB2:</td>
<td>$\text{xxx} \odot \text{RecDel}$</td>
<td>true</td>
</tr>
<tr>
<td>Reader2:</td>
<td>Reading $([[]])]$</td>
<td></td>
</tr>
<tr>
<td>Sender2:</td>
<td>$\text{lsr} \odot \text{Sending} ((m, [[]]))$</td>
<td></td>
</tr>
<tr>
<td>Unit2.status:</td>
<td></td>
<td>R</td>
</tr>
</tbody>
</table>

The whole 2WTM is stuck: TWB1 is trying to synchronise on rrr, while TWB2 is waiting to synchronize on rrr.

The situation is also revealed by studying the the observations only. The computed reduced observation state space ROSS is visualised in figure 7.9. The observation state space is computed from the reduced total state space of 366 states by application of the $\nu$-elimination algorithm based on the theory in section 6.2 of chapter 6. The full observation state space, including the MOSCA state values and value parts would contain $3m^2 + 7m + 4$ states. The difference between the reduced total state space of the abstract model and the reduced observational state space of the refined model can be explained by the added functionality at the refined model concerning the fairness constraint.
Figure 7.9: Reduced observational state space refined model

Again the graph contains final states, symbolising terminated behaviour. Each computed \( \epsilon \)-closure that contains the original final state of the total state space causes the associated state in the observational state space to become final.

A quick inspection of the \textit{Idle} execution state of the \textit{TWB} agent guided by the action sequence on the path to the final state in the spanning tree of the total state space graph reveals the problem clearly. The synchronization \texttt{rsr} is superfluous and causes the deadlock. To correct the specification the action must be deleted, resulting in the following corrected specification.

\[
\text{Idle} \triangleq \\
\text{(if status = S} \\
\text{ then (lssr } \odot \text{ rrr } \odot \text{ ColSnd } \langle[[\]} \rangle) \oplus \\
\text{(rrr } \odot \text{ lssr } \odot \text{ ColSnd } \langle[\]} \rangle) \\
\text{else (lrr } \odot \text{ rrr } \odot \text{ RecDel}) \\
\text{srec } \odot \sigma(\text{pendingSR: = true}) \odot \text{Idle}
\]

The new total state space contains 342 states connected by the same number of 1411 transitions, now without any final states. The observational state space graph is identical to the original version in figure 7.9 without the final states.
Until now I have shown that \( 2WTM \) will not deadlock due to states that have no outgoing actions at all. Another source of deadlock may be found in states with a single outgoing action which may be disabled due to the value of the MOSCA state elements, or in general states with \( k \) outgoing actions such that all \( k \) actions are disabled. To investigate this deadlock possibility it suffices to inspect the observation state space and to show for each state with \( k \) actions that \( l \) actions may be disabled, but then it should hold that \( k > l \). In the case of \( 2WTM \) it suffices to show that it is not possible to have both \texttt{send1} and \texttt{receive1} and \texttt{send2} and \texttt{receive2} disabled at the same time, which follows directly from the MOSCA specifications.

The second strategy to locate deadlock situations is to study the effect of absence of (certain) external events. Studying the specification reveals that absence of client activity is another source of stagnation of activities. The buffer switches from S-R to R-S position only after completion of precisely one transmission. Two different cases may occur.

1. In the initial situation the reading side wants to change to become sender and the sender side remains passive. Then the reduced observation state space collapses into a two state graph.

2. After a series of \textit{completed} transmissions over the S-R switch is the sending queue of waiting messages empty. The sender at the reading side of the buffer wants to send and no new messages arrive at the current sending side. Then the reduced observation state space graph remains equal to the graph depicted in figure 7.9, with again state 4 and state 5 as final states, holding the deadlocked states from the total state space resulting from the inhibited switch.

Again this fact marks a flaw in the refined model of the \( 2WTM \) as it disagrees with requirement REQ-6. A single synchronization between the two \( TWB \)'s without any additional communication appears to be not sufficient to cover the safety requirement of REQ-6.

The switch action was realized after the completion of a transmission between the two \( TWB \)'s. Another obvious point to switch is before the start of a transmission between the two buffers. This approach leads to the model given in specification 7.2

There is still another error in the specification. The model of the switch function relies on the next liveness property: a \texttt{sreq} synchronization on request of the sending agent will eventually take place. But the switch function is initiated by a synchronization, at the sender agent embedded in a conditional behaviour expression, at the \textit{TWB} part of a choice expression. As the choice in MOSCA is nondeterministic, it may not be the case that the
7.2 Revised TWB

\[ TWB \; (\text{stat}) \triangleq \begin{cases} \text{if } \text{stat} = S & \text{then } S-BufStart \\ & \text{else } R-BufStart \end{cases} \]

\[ S-BufStart \triangleq lsr \otimes mcu \otimes ColSnd \; ([]) \oplus \]
\[ \quad \text{switch} \otimes \sigma(\text{status} = R) \otimes R-BufStart \]

\[ R-BufStart \triangleq \begin{cases} \text{if } \text{pendingSR} & \text{then } \text{switch} \otimes \\ & \quad \sigma(\text{status} = S; \text{pendingSR} = \text{false}) \otimes S-BufStart \\ & \text{else } mcu \otimes lrr \otimes RecDel \end{cases} \]

\[ ColSnd \; (s) \triangleq (\text{get}(c) \otimes \text{let } \text{nbuf} = s \setminus [c] \text{ in} \]
\[ \quad \text{if } \text{len } \text{nbuf} = \text{MaxLength} \]
\[ \quad \text{then } \text{snd}(\text{pack-SPDU}(\text{nbuf})) \otimes \text{ColSnd} \; ([]) \]
\[ \quad \text{else } \text{ColSnd} \; (\text{nbuf}) \; ) \oplus \]
\[ \quad \text{eos} \otimes \text{snd}(s) \otimes S-BufEnd \]

\[ S-BufEnd \triangleq \text{cont} \otimes S-BufStart \oplus \]
\[ \quad \text{switch} \otimes \sigma(\text{status} = R) \otimes R-BufStart \]

\[ RecDel \triangleq \text{rec}(s) \odot \text{if } \text{len } \text{buf} < \text{MaxLength} \]
\[ \quad \text{then } \text{DNB} \; (\text{(unpack-SPDU}(s), \text{true})) \]
\[ \quad \text{else } \text{DNB} \; (\text{(unpack-SPDU}(s), \text{false})) \]

\[ DNB \; (\langle s, \text{last block} \rangle) \triangleq \begin{cases} \text{if } s = [] & \text{then } \text{if } \text{last block} \\ & \quad \text{then } eor \otimes R-BufEnd \\ & \text{else } RecDel \end{cases} \]
\[ \quad \text{else } \text{put}(\text{hd } s) \odot DNB \; ((\text{tl } s, \text{last block})) \]

\[ R-BufEnd \triangleq \begin{cases} \text{if } \text{pendingSR} & \text{then } \text{switch} \odot \\ & \quad \sigma(\text{status} = S; \text{pendingSR} = \text{false}) \otimes S-BufStart \\ & \text{else } \text{cont} \otimes R-BufStart \end{cases} \]
synchronization \texttt{sreq} is ever selected, given that the sending agent at the other \textit{TWB} is continuously requesting to send messages. To ensure that the lifeness property will hold another approach must be taken. The \textit{TWB} must be alerted in another way of the pending request to switch. This leads to the following adaptations.

1. The \textsc{mosca} state value \texttt{pendingSR} request is moved from the \textit{TWB} to the surrounding \textit{Unit} agent and the synchronization \texttt{sreq} is deleted completely.

2. Each function of the \textit{TWB} is given a dedicated start and stop action sequence. \textit{S-BufStart} opens the actions for the \textit{TWB} in sending mode, \textit{R-BufStart} initializes the reading mode. The switch is realized in both start and stop sequences.

   - In the start sequence the sender buffer either initializes the sending on request of the connected sending agent. \texttt{mcu} models the synchronization "message coming up" that signals the reading buffer to prepare for reception. If the reading buffer is waiting to switch the switch takes place by starting the reading buffer start sequence.
   - The reader's start sequence either offers the switch action or it responds by starting the reception.
   - The stop sequences for both sender and reader are unchanged with respect to the former model, except for a renaming.

### 7.2.6 Conclusion

The concrete \textit{2WTM} specification forms a starting point for the realization of the transmission medium build on only two simple primitives: send a block of bytes and receive a block of bytes over a singular bi-directional connection medium that is assumed to be infallible. To connect the specification to its lower levels of refinement only the two actions \texttt{snd} and \texttt{rec} of the \textit{TWB} should be addressed.

This case presentation has illustrated various aspects of working with \textsc{mosca} and revealed specific facts associated with specification in \textsc{mosca}. Some points of interest are listed below.

- \textsc{mosca}'s composition, restriction and relabeling operators enable parallel decomposition, which can be easily combined with refinement of the specification at the same time.
7.2. A communication protocol

- The need to specify on each composition the associated restriction sets and relabelling maplets is cumbersome and easily done in an erroneous way. A syntactic shorthand (like e.g. applied in LOTOS) may be advantageous here.

- To model certain aspects by behavioural operators alone may cause problems due to the combination of fairness properties and non-deterministic choices. Here state-based MOSCA specifications come to help solve this problem nicely.

The results from the analysis is summarized next.

- Carefull analysis of the protocol by the application of scenario's did not prevent the deadlock situations in the protocol. The deadlock was not introduced by transforming the scenarios into MOSCA agents but was already in the original scenario's. Their malfunction became present only after combining the scenario's through parallel composition in MOSCA. The property of deadlock freeness is certainly not compositional. Given two agents A and B and the predicate \( DF \) that signals absence of deadlock then \( DF(A) \land DF(B) \) does not guarantee \( DF(A \mid B) \).

- The operational semantics of MOSCA forms a practical basis for formal verification techniques. The verification process itself is dependent on the applied technique. The strategies applicable to state spaces are simple, but may be unpractical due to the enormous size of the investigated state spaces. Even the computation of observational state spaces only does not offer much help here, as the whole total state space is needed to construct the observational state space.

The strategies applicable to logic verification seem to me less simple, but may be associated with smaller search spaces. Further experiments and theoretical study is needed here to reveal the particular pro's and contra's of applying logics for the analysis of MOSCA specifications.

The trace analysis, although compositional in nature, offers severe problems caused by the number of traces involved. Further it is difficult to analyse properties by absence of traces.

The operational semantics of MOSCA is strong enough to enable full formal checking of all possible safety, liveness or fairness criteria. The limiting factor is rather the establishment of all possible criteria that apply to a specification. It may be the case that an important criterion is forgotten that would invalidate the specification. Automatic generation of the criteria would be certainly be of great value to the developer of the specifications.
7.3 Process control systems

This second case demonstration is placed in a total different setting as the first case demonstration. MOSCA is used as a tool to model a set of components working together to achieve a common purpose. These components comprise both hardware and software components. Section 7.3.1 opens with a short characterization of process control systems. Section 7.3.2 introduces the case demonstration. It describes the informal requirements of a home heater installation featuring an oil-burned furnace. The requirements are taken from [3] and describe a typical example of a real-time embedded software system that controls the behaviour of a surrounding system. In section 7.3.3 The controlled process is specified and section 7.3.4 presents the controlling process. The specified system is analysed in 7.3.5. The section is closed with some observations with respect to the applied method to develop the specifications and offers some comments on the MOSCA notation.

7.3.1 Introduction

A system can theoretically be described by a functional relation between inputs, outputs and time. Systems are of course not restricted to include computers, but also electro-mechanical systems, biological systems, social-economical systems etc. can be organised such that a functional relation can be taken as a description. Process control systems are often found in embedded distributed settings, like in aircraft, ships, spacecraft, manufacturing, plant control etc.

Process control is a time-based concept ([109]. The objective is to maintain a certain system property over time, during which the behaviour of the system may be influenced not only by the control system but also by the surrounding environment. This setting amounts to the classical process control loop as visualised in figure 7.10.

The behaviour of the process is controlled through the manipulated variables and the actual behaviour is monitored through the controlled variables. The actuators are devices that manipulate the behaviour of a process by adjusting the manipulated variables, such as a valve regulating the flow of some fluid. The sensors monitor the controlled variables of the process, e.g. a barometer that registers the pressure in a vat. The process is also influenced by disturbances not subject to adjustment or control.

The main disadvantage of the mathematical approach is the extreme level of abstraction needed to enable the construction of a computable function ([140], [141]). Often these processes are non linear, discontinuous and highly irregular. The obtained mathematical description is certainly of
Figure 7.10: Basic model of a process and its control system

value to analyse the overall function of a process and can be used to compute parameters of the process variables, but it alone is not a solid base for the design of the system. In the next sections I will show how MOSCA can be used to record the requirements of a process control system such that they will form a good starting point for the design of the components of the system.

7.3.2 The heater

Consider a simple heater system, for which the informal requirements are given in figure 7.3.2. Although this description is to be considered far from complete, as pointed out by Wood in [223], it illustrates the specific real-time behaviour of reactive systems.

Taking into account that the physical components, the heater system is built from, are known, some object-oriented paradigm seems reasonable here.

From the informal requirements description a set of physical objects can be identified. The main function of the controller is to control their behaviour.
The controller of an oil hot water home heating system regulates in-flow of heat, by turning the furnace on and off, and monitors the status of combustion and fuel flow of the furnace system, provided the master switch is set to HEAT position. The controller activates the furnace whenever the home temperature $t$ falls below $t_r - 2$ degrees, where $t_r$ is the desired temperature set by the user. The activation procedure is as follows:

1. the controller signals the motor to be activated.

2. the controller monitors the motor speed and once the speed is adequate it signals the ignition and oil valve to be activated.

3. the controller monitors the water temperature and once the temperature is reached a predefined value it signals the circulation valve to be opened. The heater water then starts to circulate through the house.

4. a fuel flow indicator and an optional combustion sensor signal the controller if abnormalities occur. In this case the controller signals the system to be shut off.

5. once the home temperature reaches $t_r + 2$ degrees, the controller deactivates the furnace by first closing the oil valve and then, after 5 seconds, stopping the motor.

Further the system is subject to the following constraints:

1. minimum time for furnace restart after prior operation is 5 minutes.

2. furnace turn-off shall be indicated within 5 seconds of master switch shut off or fuel flow shut off.

Figure 7.11: Informal requirements of Heating System
7.3. Process control systems

7.3.2.1 A taxonomy of real-world objects

The taxonomy is restricted to the functional objects. All connecting circuits between the objects such as the electric circuits, the water conducting pipes etc. are assumed present. The following list is derived from the informal description of the system.

1. The furnace. The furnace is composed of: a fuel valve regulating the fuel flow from a fuel vat, an ignition device, a furnace status indicator lamp, a fuel flow sensor, a combustion sensor.

2. The water circulation system. The water circulation system consists of: a water valve regulating the water flow in a closed circuit, a motor.

3. The master switch.

4. The home in which the heater functions.

5. The control system itself.

7.3.2.2 The actuators and associated manipulated variables

<table>
<thead>
<tr>
<th>Actuator</th>
<th>Associated manipulated variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel valve</td>
<td>quantity of fuel arriving in the furnace for combustion</td>
</tr>
<tr>
<td>Ignition device</td>
<td>presence/absence of fuel combustion</td>
</tr>
<tr>
<td>Furnace Lamp</td>
<td>presence/absence of furnace activity</td>
</tr>
<tr>
<td>Water valve</td>
<td>the active water circuit: either inner furnace circulation or home circulation</td>
</tr>
<tr>
<td>Motor</td>
<td>presence/absence of activity or water flow</td>
</tr>
</tbody>
</table>

The state of the actuators is modelled as follows.

<table>
<thead>
<tr>
<th>Actuator</th>
<th>MOSCA model of actuator state</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuel valve</td>
<td>OPEN</td>
</tr>
<tr>
<td>Ignition device</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>Furnace Lamp</td>
<td>ON</td>
</tr>
<tr>
<td>Water valve</td>
<td>OPEN</td>
</tr>
<tr>
<td>Motor</td>
<td>PASSIVE</td>
</tr>
</tbody>
</table>
7.3.2.3 The sensors and associated controlled variables

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Controlled variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor speed sensor</td>
<td>motor speed</td>
</tr>
<tr>
<td>Water temperature sensor</td>
<td>water temperature</td>
</tr>
<tr>
<td>Fuel flow sensor</td>
<td>monitors the status of the fuel flow</td>
</tr>
<tr>
<td>Combustion sensor</td>
<td>monitors the status of the combustion of fuel</td>
</tr>
<tr>
<td>Home temperature sensor</td>
<td>home temperature</td>
</tr>
<tr>
<td>Master switch</td>
<td>presence/absence of activity of heater system</td>
</tr>
</tbody>
</table>

The controlled variables are modelled as follows.

<table>
<thead>
<tr>
<th>Controlled variable</th>
<th>Associated MOSCA value domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor speed</td>
<td>number of revolutions: N</td>
</tr>
<tr>
<td>Water temperature</td>
<td>degrees: Z</td>
</tr>
<tr>
<td>Fuel flow</td>
<td>amount of flow: Z</td>
</tr>
<tr>
<td>Combustion of fuel</td>
<td>TOO_LOW</td>
</tr>
<tr>
<td>Home temperature</td>
<td>degrees: Z</td>
</tr>
<tr>
<td>Master switch</td>
<td>HEATING</td>
</tr>
</tbody>
</table>

The numerical models present scaled values in some subset domain of the integer numbers. The union type literals represent specific sensor actions.

7.3.3 The heater process

The heater process realizes the top part of the process control system. In

![Figure 7.12: Basic model of the process part](image)

the following presentation of the heater system all physical objects are, in
contrast to the former case description not assumed infallible: they may wear down! This assumption models a realistic setting of the heater specification, but creates a far more complex specification. Our interpretation of the heater system is built from the following objects.

7.3.3.1 The room

The model for the room can be derived, starting from two different approaches: the physical conditions in the room change autonomously, or the experimenter is able to control the physical conditions of the room.

The autonomous room The main attribute of the physical state of the room is its temperature, visualised through the device that registrates it, a thermometer. Linked to the thermometer is an event generator that signals whenever a rising temperature reaches $t_{\text{max}} + 2$ degrees or signals whenever a falling temperature reaches $t_{\text{min}} - 2$ degrees, where $t_{\text{max}}$ is a preset value that resembles the maximum room temperature that is acceptable. The dual value $t_{\text{min}}$ holds the minimum room temperature that is acceptable. These two constants are attributes of the event generator. In the case the event generator is only able to register cyclic rising and falling crossings of the maximum and minimum values, the model may be expressed as follows.

$$ROOM_1$$
$$\text{ports syn room\_too\_cold}$$
$$\text{syn room\_too\_hot}$$

$$ROOM_1 \triangleq$$
$$\text{idle(cooling\_down()) \circ room\_too\_cold \circ}$$
$$\text{idle(warming\_up()) \circ room\_too\_warm \circ ROOM_1}$$

It features a function $warming\_up()$ that may realize a mathematical approximation of the physical effect of warming up the room due to whatever physical source. The dual function $cooling\_down()$ reflects the opposite effect of cooling down the room.

In the case that the event generating device can register any crossing of the extreme values the model looks slightly different. Now the room can become repeatedly too hot or too cold. The MOSCA realization features the choice operator.

$$ROOM_2$$
$$\text{ports syn room\_too\_cold}$$
$$\text{syn room\_too\_hot}$$
\[ \text{ROOM}_2 \triangleq \\
idle(\text{sensing}) \circ \\
(room\_too\_cold \circ \text{ROOM}_2) \circ \\
(room\_too\_warm \circ \text{ROOM}_2) \]

The \text{sensing} function models the passing of time between two temperature crossings of the extremes, either maximum or minimum.

**Experimenter controlled room** In the case the experimenter controls the event generating device the behaviour of the room is dictated by external events completely. A MOSCA realization is given next.

\[ \text{ROOM}_3 \\
\text{ports syn set\_room\_too\_cold} \\
\text{syn room\_too\_cold} \\
\text{syn set\_room\_too\_warm} \\
\text{syn room\_too\_warm} \\
\text{ROOM}_3 \triangleq \\
(set\_room\_too\_cold \circ \text{room\_too\_cold} \circ \text{ROOM}_3) \circ \\
(set\_room\_too\_warm \circ \text{room\_too\_warm} \circ \text{ROOM}_3) \]

### 7.3.3.2 The main switch

The physical appearance of the main switch may take may possible forms. Most of all different realizations of the switch fall in two classes marked as the "on/off push button model" and the "tumble model". Both models are elaborated upon.

**On/off push button model** The switch consists of two buttons, one marked \text{ON} and one marked \text{OFF}. Pushing the \text{ON} button results in a signal to the heating system only if the main switch was in \text{OFF} position. Pushing the \text{OFF} button results in a shut-down signal to the heater control only if the main switch was in the \text{ON} position.

\[ \text{MAIN\_SWITCH\_PB} \\
\text{ports syn ms\_on} \\
\text{syn ms\_off} \\
\text{syn ms\_on\_event} \\
\text{syn ms\_off\_event} \\
\text{state status : \{ACTIVE, INACTIVE\}} \]
7.3. Process control systems

Figure 7.13: Main switch push-button model

\[
\text{init status } \triangleq \text{ status } = \text{INACTIVE}
\]

\[
\text{MAIN SWITCH PB } \triangleq
\]

\[
\quad (\text{ms on } \odot \text{ if status } = \text{ACTIVE}) \quad \begin{align*}
\text{then } & \text{MAIN SWITCH PB} \\
\text{else } & \sigma(\text{status: } = \text{ACTIVE}) \odot \\
& \text{ms on event } \odot \text{MAIN SWITCH PB} \oplus
\end{align*}
\]

\[
\quad (\text{ms off } \odot \text{ if status } = \text{INACTIVE}) \quad \begin{align*}
\text{then } & \text{MAIN SWITCH PB} \\
\text{else } & \sigma(\text{status: } = \text{INACTIVE}) \odot \\
& \text{ms off event } \odot \text{MAIN SWITCH PB}
\end{align*}
\]

The tumble model. In the tumble model the switch its two physical states are directly connected to the operational parts of the switch (fig. 7.14). The switch nob can either be in the OFF position or in the ON position.

\[
\text{MAIN SWITCH T}
\]

ports syn tumble

\[
\quad \text{syn ms on event}
\]

\[
\quad \text{syn ms off event}
\]

state status : \{\text{ACTIVE, INACTIVE}\}

init status \triangleq \text{ status } = \text{INACTIVE}
Figure 7.14: Main switch tumble model

\[ \text{MAIN\_SWITCH\_T} \triangleq \]
\[ \text{tumble} \odot \text{if status} = \text{INACTIVE} \]
\[ \text{then } \sigma(\text{status} = \text{ACTIVE}) \odot \]
\[ \text{ms\_on\_event} \odot \text{MAIN\_SWITCH\_T} \]
\[ \text{else } \sigma(\text{status} = \text{INACTIVE}) \odot \]
\[ \text{ms\_off\_event} \odot \text{MAIN\_SWITCH\_T} \]

The MOSCA realization models a switch that will always be in a standard initial state.\(^3\)

7.3.3.3 The heater-unit

The heater-unit consists of the water circulation system and the furnace. It regulates the water flow, controls the motor, fuel valve and furnace light. The motor accepts signals to either start or to stop running. It indicates the outer world that a certain speed is reached. The oil valve and furnace light, both server objects accept signals to be opened or closed, and to go on and off.

The motor The imagined motor features a starter, and signals its environment that it has reached working speed. Once active it can be stopped.

\[ \text{MOTOR} \]
\[ \text{ports syn start\_motor} \]
\[ \text{syn motor\_running} \]
\[ \text{syn stop\_motor} \]

---

\(^3\)This reflects a strict constraint on the assemblage of the heater in reality. The switch is always be mounted in the OFF position.
7.3. Process control systems

\[
\text{MOTOR} \triangleq \\
\text{start\_motor} \circ \text{idle}(\text{motor\_startup\_time}) \circ \\
\text{motor\_running} \circ \text{stop\_motor} \circ \text{MOTOR}
\]

The time the motor takes to reach its working speed is computed by the function \text{motor\_startup\_time}().

The water valve, fuel valve and furnace light These three devices all have a binary state. The water and fuel valve are either open or closed. During mounting these devices are closed. The furnace light is either off or on. The MOSCA specifications reflect the simple cyclic behaviour of these devices. Each device contains a specific mechanism that, whenever activated, takes a certain amount of time to complete its action. E.g. to open the water valve takes a constant amount of time, defined by the constant \text{water\_v\_opening\_time}.

\[
\text{WATER\_VALVE}
\]
\[
\text{ports syn open}
\]
\[
\text{syn close}
\]

\[
\text{WATER\_VALVE} \triangleq \\
\text{open} \circ \text{idle} \text{water\_v\_opening\_time} \circ \\
\text{close} \circ \text{idle} \text{water\_v\_closing\_time} \circ \text{WATER\_VALVE}
\]

\[
\text{FUEL\_VALVE}
\]
\[
\text{ports syn open}
\]
\[
\text{syn close}
\]

\[
\text{FUEL\_VALVE} \triangleq \\
\text{open} \circ \text{idle} \text{fuel\_v\_opening\_time} \circ \\
\text{close} \circ \text{idle} \text{fuel\_v\_closing\_time} \circ \text{FUEL\_VALVE}
\]

\[
\text{FURNACE\_LIGHT}
\]
\[
\text{ports syn on}
\]
\[
\text{syn off}
\]

\[
\text{FURNACE\_LIGHT} \triangleq \\
\text{on} \circ \text{idle} \text{furnace\_l\_on\_time} \circ \\
\text{off} \circ \text{idle} \text{furnace\_l\_off\_time} \circ \text{FURNACE\_LIGHT}
\]
The water temperature sensor  The water temperature is used to control the water valve. The device is equivalent to the event generator situated in the room (although perhaps constructed totally different). It holds two constants that controls the event generator: $t_{wout}$ and $t_{umin}$. It generates two signals: $wout$ and $water\_too\_cold$. The measuring device features a moving current temperature pointer (ctp) that moves along the temperature scale. If the ctp crosses one of the extrema, the associated event is signalled.

\[
WATER\_TEMP\_SENSOR
\]
ports syn $wout$

\[
syn water\_too\_cold
\]

shares $WATER\_COOLING$, $WATER\_RISING$

\[
WATER\_TEMP\_SENSOR \triangleq
\]
idle$(sensing()) \odot
\]

\[
wout \odot WATER\_COOLING \oplus
\]

\[
water\_too\_cold \odot WATER\_RISING
\]

\[
WATER\_COOLING \triangleq
\]
idle$(sensing()) \odot water\_too\_cold \odot WATER\_RISING
\]

\[
WATER\_RISING \triangleq
\]
idle$(sensing()) \odot wout \odot WATER\_COOLING
\]

The ignition device  The ignition device offers a single capability, once activated: producing an electric spark.

\[
SPARK
\]
ports syn $spark$

\[
SPARK \triangleq
\]

\[
spark \odot SPARK
\]

The fuel flow sensor  This device measures mechanically the quantity of fuel that passes its sensor. It is operational whenever the furnace is activated. It continuously holds its most recent measurement. The device can be read out on demand.

\[
FUEL\_FLOW\_SENSOR
\]
ports out read\_out : Voltage
7.3. Process control systems

\[
\text{FUEL.FLOWSENSOR} \triangleq \\
\text{read.out(fuel.flow())} \odot \text{FUEL.FLOWSENSOR}
\]

The combustion sensor  This device, once activated monitors the combustion in an autonomous way. It holds e.g. a small bi-metal core that registers the heat effect of the combustion. Whenever the combustion generates too much heat, the bi-metal fires. Whenever the combustion generates too less heat, another measuring device generates a warning signal.

\[
\text{COMBUSTIONSENSOR} \\
\text{ports syn combustion.too.low} \\
\text{syn combustion.too.high} \\
\text{syn combustion.going}
\]

\[
\text{COMBUSTIONSENSOR} \triangleq \\
\text{idle(sensing())} \odot \\
\text{combustion.too.low} \odot \text{COMBUSTIONSENSOR} \odot \\
\text{combustion.too.high} \odot \text{COMBUSTIONSENSOR} \odot \\
\text{combustion.going} \odot \text{COMBUSTIONSENSOR}
\]

7.3.3.4 Process assembly

The whole process now is assembled from two main components: the house in which the heater functions and the heater itself. The house is represented by the main switch and the room in which the temperature sensor is active. The selected components are e.g. the second ROOM model and a tumble main switch. First the HOUSE agent is specified.

\[
\text{HOUSE} \\
\text{ports syn MS.tumble, MS.ms.on.event, MS.ms.off.event} \\
\text{syn R.room.too.cold, R.room.too.warm}
\]

\[
\text{HOUSE} \triangleq \text{let } MS = \text{MAIN_SWITCH.T} \\
\text{[tumble } \mapsto \text{MS.tumble,} \\
\text{ms.on.event } \mapsto \text{MS.ms.on.event,} \\
\text{ms.off.event } \mapsto \text{MS.ms.off.event]} \\
R = \text{ROOM.2} \\
\text{[room.too.cold } \mapsto \text{R.room.too.cold,} \\
\text{room.too.hot } \mapsto \text{R.room.too.hot]} \\
\text{in MS} \mid R
\]
The constituents of the house are assembled in the composition in an instantiation with private port names. This technique enables co-existence of various copies of the same agent in one composition. In the other parts of the process specification the same technique is applied.

The heater is assembled from two components: the water subsystem and the furnace subsystem.

\[
\text{HEATER}
\]

\[
\text{HEATER} \triangleq \text{FURNACE\_SUBSYSTEM} | \text{WATER\_SUBSYSTEM}
\]

The furnace subsystem features the fuel valve, furnace light, ignition device, fuel flow sensor and combustion sensor.

\[
\text{FURNACE\_SUBSYSTEM}
\]
7.3. Process control systems

\[ FURNACE\_SUBSYSTEM \triangleq \]
\[
\text{let } FV = FUEL\_VALVE \\
\quad [\text{open} \leftrightarrow FV\_open, \text{close} \leftrightarrow FV\_close] \\
FL = FURNACE\_LIGHT \\
\quad [\text{on} \leftrightarrow FL\_on, \text{off} \leftrightarrow FL\_off] \\
SP = SPARK \\
\quad [\text{spark} \leftrightarrow SP\_spark] \\
FFS = FUEL\_FLOW\_SENSOR \\
\quad [\text{read}\_out \leftrightarrow FFS\_read\_out] \\
CS = COMBUSTION\_SENSOR \\
\quad [\text{combustion\_too\_low} \leftrightarrow CS\_combustion\_too\_low, \\
\quad \text{combustion\_too\_high} \leftrightarrow CS\_combustion\_too\_high, \\
\quad \text{combustion\_going} \leftrightarrow CS\_combustion\_going] \\
FURNACE\_ACTORS = FV \mid FL \mid SP \\
FURNACE\_SENSORS = FFS \mid CS \\
in FURNACE\_ACTORS \mid FURNACE\_SENSORS
\]

\[ WATER\_SUBSYSTEM \]
ports syn M.start.motor \\
syn M.motor.running \\
syn M.stop.motor \\
syn WV.open \\
syn WV.close \\
syn WTS.wowt \\
syn WTS.water\_too\_cold

\[ WATER\_SUBSYSTEM \triangleq \]
\[
\text{let } M = MOTOR \\
\quad [\text{start}\_motor \leftrightarrow M\_start.motor, \\
\quad \text{motor}\_running \leftrightarrow M\_motor\_running, \\
\quad \text{stop}\_motor \leftrightarrow M\_stop.motor] \\
WV = WATER\_VALVE \\
\quad [\text{open} \leftrightarrow WV\_open, \\
\quad \text{close} \leftrightarrow WV\_close] \\
WTS = WATER\_TEM\_SENSOR \\
\quad [\text{wowt} \leftrightarrow WTS\_wowt, \\
\quad \text{water\_too\_cold} \leftrightarrow WTS\_water\_too\_cold] \\
WATER\_ACTORS = M \mid WV \\
WATER\_SENSORS = WTS \\
in WATER\_ACTORS \mid WATER\_SENSORS
\]

Finally the process unit itself becomes:
**PROCESS**

ports syn MS.tumble
    syn MS.ms_on_event
    syn MS.ms_off_event
    syn R.room_too_cold
    syn R.room_too_warm
    syn FV.open, FV.close
    syn FL.on, FL.off
    syn SP.spark
    out FFS.read_out : N
    syn CS.combustion_too_low
    syn CS.combustion_too_high
    syn CS.combustion_going
    syn M.start_motor
    syn M.motor_running
    syn M.stop_motor
    syn WV.open
    syn WV.close

**PROCESS ⊆ HOUSE | HEATER**

### 7.3.4 The control component

The control component consists of three components: the interface to the actuators, the interface to the sensors and the controlling component itself. The control component represents the lower part of figure 7.10. The spec-

![Figure 7.15: Basic model of the control part](image)

The specification of the controller may be designed along different paradigms, such as state based paradigms or a functional continuation paradigms. In the
state based approach a MOSCA state concept is managed within the controller that reflects the active state of the heater and the house during the lifetime of the involved agents. In [213] another specification of the heater is presented that was based on the functional continuation approach. Each state of the system is represented by a specific execution state, i.e. a piece of MOSCA code. All the execution states were linked by certain actions that modelled state changes. Thus the overall specification strongly resembled a finite state machine. In this presentation the state based approach is taken.

7.3.4.1 The interface to the actuators

The house agent offers no actuators. The heater agent offers actuators in both subsystems. The interface to the actuators is formed by the set of actions that model operations to the actuators. The interfacing devices are modelled by a simple latch that passes on the action to their destination. The interface agents model the real-world interfaces between the controller and the actuators. To avoid unnecessary complication of the specification it is assumed that the actuators accept their commands from the interface devices even during malfunctioning. A direct result of this assumption is found in the design of the controller later on. Malfunctioning of certain actuators must be observed through exceptional behaviour of the system, rather through direct timing out of the interface commands. The interface to the actuators in the furnace subsystem consists of:

\[ FURNACE\_AI \]
\[
\text{ports syn FV.open, FV.close}
\]
\[
\text{syn FVI.open, FVI.close}
\]
\[
\text{syn FL.on, FL.off}
\]
\[
\text{syn FLI.on, FLI.off}
\]
\[
\text{syn SP.spark}
\]
\[
\text{syn SPI.spark}
\]

\[ FURNACE\_AI \triangleq \]
\[
\text{FVI.open } \odot \text{ idle } 1 \odot \text{ FV.open } \odot \text{ FURNACE\_AI } \odot
\]
\[
\text{FVI.close } \odot \text{ idle } 1 \odot \text{ FV.close } \odot \text{ FURNACE\_AI } \odot
\]
\[
\text{FLI.on } \odot \text{ idle } 1 \odot \text{ FLI.on } \odot \text{ FURNACE\_AI } \odot
\]
\[
\text{FLI.off } \odot \text{ idle } 1 \odot \text{ FLI.off } \odot \text{ FURNACE\_AI } \odot
\]
\[
\text{SPI.spark } \odot \text{ idle } 1 \odot \text{ SP.spark } \odot \text{ FURNACE\_AI}
\]

The interface to the actuators in the water subsystem consists of:
WATER_AI

ports syn M1.start.motor
    syn M1.start.motor
    syn M1.stop.motor
    syn M1.stop.motor
    syn WV1.open
    syn WV.open
    syn WV1.close
    syn WV.close

WATER_AI =

M1.start.motor @ idle1 @ M1.start.motor @ WATER_AI @
M1.stop.motor @ idle1 @ M1.stop.motor @ WATER_AI @
WV1.open @ idle1 @ WV.open @ WATER_AI @
WV1.close @ idle1 @ WV.close @ WATER_AI @

7.3.4.2 The interface to the sensors

Sensory devices fall in two major subclasses: devices that generate events whenever they have measured exceptional behaviour and devices that are polled to deliver their measurements. E.g. the device in the room is an event generating sensory device. To present the MOSCA model of a polled device I have collected such device in the heater specification: the fuel flow indicator. I have chosen to specify each interface device to a polled sensor separately. The polling client is setting the polling specification. It is not possible to specify nested actions, so for each poll two actions are necessary: one to active the poll and one to achieve the result. 4

FURNACE_SI

ports in FFSI.read_out : N
    syn FFSI.poll
    out FFSI.read_out : N
    syn CSI.combustion_too_low
    syn CSI.combustion_too_low
    syn CSI.combustion_too_high
    syn CSI.combustion_too_high
    syn CSI.combustion_going
    syn CSI.combustion_going

---

4This e.g. in contrast to the rendezvous concept such as applied in Ada, where the rendezvous concept may be applied nested.
7.3. Process control systems

\[ \text{FURNACE}_\text{SI} \triangleq \]
\[ \text{let } \text{FURNACE}.\text{POLLING}_\text{HANDLERS} = \text{FFSI}_\text{INTERFACE} \]
\[ \text{in } \text{FURNACE}_\text{IH} \mid \text{FURNACE}.\text{POLLING}_\text{HANDLERS} \]

\[ \text{FFSI}_\text{INTERFACE} \triangleq \]
\[ \text{ports in FFS}.\text{read}_\text{out} : \mathbb{N} \]
\[ \text{syn FFSI}.\text{poll} \]
\[ \text{out FFS}.\text{read}_\text{out} : \mathbb{N} \]

\[ \text{FFSI}_\text{INTERFACE} \triangleq \]
\[ \text{FFSI}.\text{poll} \circ \text{FFS}.\text{read}_\text{out(}\text{fuel}_\text{flow}) \circ \]
\[ \text{FFS}.\text{read}_\text{out(}\text{fuel}_\text{flow}) \circ \text{FFSI}_\text{INTERFACE} \]

\[ \text{FURNACE}_\text{IH} \]
\[ \text{ports syn CSI}.\text{combustion}_\text{too}_\text{low} \]
\[ \text{syn CSI}.\text{combustion}_\text{too}_\text{low} \]
\[ \text{syn CSI}.\text{combustion}_\text{too}_\text{high} \]
\[ \text{syn CSI}.\text{combustion}_\text{too}_\text{high} \]
\[ \text{syn CSI}.\text{combustion}_\text{going} \]
\[ \text{syn CSI}.\text{combustion}_\text{going} \]

\[ \text{FURNACE}_\text{IH} \triangleq \]
\[ \text{CSI}.\text{combustion}_\text{too}_\text{low} \circ \text{idle} \circ \]
\[ \text{CSI}.\text{combustion}_\text{too}_\text{low} \circ \text{FURNACE}_\text{IH} \circ \]
\[ \text{CSI}.\text{combustion}_\text{too}_\text{high} \circ \text{idle} \circ \]
\[ \text{CSI}.\text{combustion}_\text{too}_\text{high} \circ \text{FURNACE}_\text{IH} \circ \]
\[ \text{CSI}.\text{combustion}_\text{going} \circ \text{idle} \circ \]
\[ \text{CSI}.\text{combustion}_\text{going} \circ \text{FURNACE}_\text{IH} \]

The water subsytem sensor interface is modelled as

\[ \text{WATER}_\text{SI} \]
\[ \text{ports syn MI}.\text{motor}_\text{running} \]
\[ \text{syn M}.\text{motor}_\text{running} \]
\[ \text{syn WTS}.\text{wo}t \]
\[ \text{syn WTS}.\text{wo}t \]
\[ \text{syn WTS}.\text{water}_\text{too}_\text{cold} \]
\[ \text{syn WTS}.\text{water}_\text{too}_\text{cold} \]
\( \text{WATER} \_ \text{SI} \triangleq \)
\[ \text{M.motor\_running} \circ \text{MI.motor\_running} \circ \]
\[ \text{id1} \circ \text{WATER} \_ \text{SI} \circ \]
\[ \text{WTS.wout} \circ \text{id1} \circ \]
\[ \text{WTSI.wout} \circ \text{WATER} \_ \text{SI} \circ \]
\[ \text{WTS.water\_too\_cold} \circ \text{id1} \circ \]
\[ \text{WTSI.water\_too\_cold} \circ \text{WATER} \_ \text{SI} \]

Finally the house sensor interface becomes:

\( \text{HOUSE} \_ \text{SI} \)
ports syn MSI.ms\_on\_event
\[ \text{syn MS.ms\_on\_event} \]
\[ \text{syn MSI.ms\_off\_event} \]
\[ \text{syn MS.ms\_off\_event} \]
\[ \text{syn RI.room\_too\_cold} \]
\[ \text{syn R.room\_too\_cold} \]
\[ \text{syn RI.room\_too\_warm} \]
\[ \text{syn R.room\_too\_warm} \]

\( \text{HOUSE} \_ \text{SI} \triangleq \)
\[ \text{MS.ms\_on\_event} \circ \text{id1} \circ \text{MSI.ms\_on\_event} \circ \text{HOUSE} \_ \text{SI} \circ \]
\[ \text{MS.ms\_off\_event} \circ \text{id1} \circ \text{MSI.ms\_off\_event} \circ \text{HOUSE} \_ \text{SI} \circ \]
\[ \text{R.room\_too\_cold} \circ \text{id1} \circ \text{RI.room\_too\_cold} \circ \text{HOUSE} \_ \text{SI} \circ \]
\[ \text{R.room\_too\_warm} \circ \text{id1} \circ \text{RI.room\_too\_warm} \circ \text{HOUSE} \_ \text{SI} \]

7.3.4.3 The state of the system

The state of the heater system and the house is recorded into the following MOSCA state construction:

state master\_switch : OFF | HEATING
room : TOO\_COLD | TOO\_HOT
motor : PASSIVE | STARTING | RUNNING
water\_valve : OPEN | CLOSED
furnace : WARMING\_UP | BURNING | OFF

The state space of the state component is constrained by causal relationships recorded in the state invariant:
7.3. Process control systems

\[
\text{inv } mk\text{-Controller}(ms, r, m, wv, f) \triangleq \\
\quad (ms = \text{OFF} \iff \\
\quad \quad (m = \text{PASSIVE} \land f = \text{OFF})) \land \\
\quad (ms = \text{HEATING} \iff \\
\quad \quad (f = \text{BURNING} \iff m = \text{RUNNING}))
\]

The state space is initialized as follows:

\[
\text{init } mk\text{-Controller}(ms, r, m, wv, f) \triangleq \\
\quad mk\text{-Controller}(\text{OFF}, -r, \text{PASSIVE}, \text{CLOSED}, \text{OFF})
\]

Next to the state the controller needs some knowledge to interpret the measured values of the fuel flow:

values
\[
\begin{align*}
\text{fuel}\_\text{flow}\_\text{setting} & : \mathbb{N} = \ldots \\
\text{fuel}\_\text{flow}\_\text{deviation} & : \mathbb{N} = \ldots
\end{align*}
\]

end

7.3.4.4 The controller

The controller agent is designed as a collection of event handlers that react on the events generated by the sensors in the system. The top level event handler is contained in the Controller agent itself.

\[
\text{Controller} \triangleq \\
\quad \text{MSI}\_\text{ms}\_\text{on}\_\text{event} \odot \sigma (\text{master}\_\text{switch} = \text{HEATING}) \odot \\
\quad \text{MSI}\_\text{ms}\_\text{off}\_\text{event} \odot \sigma (\text{master}\_\text{switch} = \text{OFF}) \odot \\
\quad \text{RI}\_\text{room}\_\text{too}\_\text{cold} \odot \sigma (\text{room} = \text{TOO}_\text{COLD}) \odot \\
\quad \text{MSI}\_\text{ms}\_\text{on}\_\text{event} \odot \sigma (\text{master}\_\text{switch} = \text{OFF}) \odot \\
\quad \text{SHUT}\_\text{DOWN}\_\text{HANDLER} \\
\quad \text{RI}\_\text{room}\_\text{too}\_\text{warm} \odot \sigma (\text{room} = \text{TOO}_\text{WARM}) \odot \\
\quad \text{SHUT}\_\text{DOWN}\_\text{HANDLER} \\
\quad \text{RI}\_\text{room}\_\text{too}\_\text{cold} \odot \\
\quad \text{STARTUP}\_\text{FURNACE} \\
\quad \text{TSI}\_\text{wout} \odot \\
\quad \text{HEATING} \\
\quad \text{WTSI}\_\text{water}\_\text{too}\_\text{cold} \odot \\
\quad \text{WATER}_\text{TOO}_\text{COLD}\_\text{HANDLER} \\
\quad \text{enabled}_\text{furnace} \odot \\
\quad \text{ENABLE}_\text{FURNACE}\_\text{HANDLER} \\
\quad \text{Timeout}_\text{furnace}\_\text{warming}_\text{up}\_\text{timer} \odot \text{Alarm} \\
\quad \text{Timeout}_\text{motor}\_\text{starting}_\text{up}\_\text{timer} \odot \text{Alarm}
\]
The first handler takes care of the `ms_on_event` generated by the sensor that monitors the main switch. Whenever the room needs heating and the furnace is not disabled and the motor is not already started due to an earlier `ms_on_event` the heater may be activated.

\[
\text{MS}_{-}\text{ON}_{-}\text{HANDLER} \triangleq \\
\text{if} \quad \text{room} = \text{TOO}_{-}\text{COLD} \land \text{furnace} \neq \text{DISABLED} \land \\
\text{motor} = \text{PASSIVE} \\
\text{then} \quad \text{STARTUP}_{-}\text{MOTOR} \\
\text{else} \quad \text{Controller}
\]

On switching off the system several cases may occur. If the heater is not active, no actions are needed. If the event occurs during the start up procedure of the motor, but before activation of the furnace, the motor is stopped. If the event occurs during a heating furnace, the furnace is shut down. No other possibilities are possible. The shut down procedure is not interruptable and results in a furnace that will remain inactive during 5 minutes.

\[
\text{SHUT}_{-}\text{DOWN}_{-}\text{HANDLER} \triangleq \\
\text{if} \quad \text{furnace} = \text{BURNING} \\
\text{then} \quad \text{SHUTDOWN (NORMAL}_{-}\text{SHUT}_{-}\text{DOWN}) \\
\text{else if} \quad \text{furnace} = \text{OFF} \land \text{motor} = \text{STARTING} \\
\text{then} \quad \text{STOP}_{-}\text{MOTOR} \\
\text{else Controller}
\]

On the event that signals the moment the room becomes too cold the furnace is activated whenever it is enabled and not already functioning or starting up.

\[
\text{ROOM}_{-}\text{TOO}_{-}\text{COLD}_{-}\text{HANDLER} \triangleq \\
\text{if} \quad \text{master}\_\text{switch} = \text{HEATING} \land \text{furnace} \neq \text{DISABLED} \land \\
\text{motor} \neq \text{starting} \\
\text{then} \quad \text{STARTUP}_{-}\text{MOTOR} \\
\text{else Controller}
\]

The next handler reacts to the cooling water in the system. If the event arrives after the furnace was shut down it falls in the natural flow of actions and nothing is done. If the water cools down during an active furnace, something is completely wrong and the alarming routine is started. It may be caused by a defect in the water sensing device.
WATER TOO COLD HANDLER
\[ \triangleq \]
if \( \text{furnace} = \text{BURNING} \)
then \text{Alarm}
else \text{Controller}

The \text{ENABLE FURNACE HANDLER} reacts on the time-out of the 5 minutes delay after shut down of the furnace. If in the mean time the room has cooled down too much, the furnace is re-activated.

ENABLE FURNACE HANDLER
\[ \triangleq \]
if \( \text{room} = \text{TOO COLD} \land \text{master switch} = \text{HEATING} \)
then \text{STARTUP MOTOR}
else \text{Controller}

The next agent expresses the motor starting routine. First the motor is activated through its actuator interface. If the water valve was open, it is closed so the water in the furnace is brought to working temperature first. A timer is started to monitor the time spent on starting up the motor. The timer will generate an interrupt after the normal starting up time is expired augmented with a safety margin \( \delta_{\text{mst}} \) (see the \text{Motor starting up timer} specification). If the controller receives the \text{motor running} event before the timer has expired, the timer is disabled and the starting up procedure may proceed.

STARTUP MOTOR \[ \triangleq \]
\[ \text{Ml start motor} \odot \]
\[ \sigma(\text{motor} := \text{STARTING}) \odot \]
(if \( \text{water valve} = \text{OPEN} \)
then \text{WVI close}
else idle 0) \odot
(\text{Controller} | \text{Motor starting up timer})

The normal starting up procedure of the motor can be interrupted by a master switch event arriving before the starting up procedure was completed. Accordingly the motor is stopped. First the motor running signal is awaited, followed by stopping the motor. To prevent repeated starting and stopping of the motor induced by erratic manipulation of the master switch the whole heater is disabled for 5 minutes.
\textit{STOP\_MOTOR} \triangleq \\
\text{MI.motor\_running} \odot \\
\text{MI.stop\_motor} \odot \\
\sigma(\text{motor:} = \text{STOPPED}) \odot \\
(\text{Controller} \mid \text{Disable\_furnace})

The furnace is activated by first opening the fuel valve. This device does not signals its completed actions. However, the fact of its failing to open can be registered by monitoring the fuel flow. Whenever the valve is opened the fuel flow will initiate. This process is monitored by sampling the fuel mass in the furnace (the \textit{Poll\_fuel\_flow\_start} agent). If the sampling is timed out by a delay before enough fuel has accumulated in the furnace the alarm routine is invoked. Next the ignition device is activated. It is assumed that the technical qualities of the device guarantee an ignition within two generated sparks separated by a predefined delay. Again a time out is used to register a possible misfiring of the ignition device. The succeeded ignition is signalled by the combustion sensor and the furnace light is activated. The heater is warming up. After the standard warming up period, in which the water in the furnace is heated the heater becomes in its working situation: the water valve is opened and the heated water starts to circulate through the house. The standard warming up time is timed-out and if the furnace fails to heat the water within the preset period the alarm routine is again invoked.

\textit{STARTUP\_FURNACE} \triangleq \\
\text{FVI.open} \odot \\
(Poll\_fuel\_flow\_start \\
(idle\_fuel\_v\_open\_time + \delta_{\text{fo}} \odot \text{Alarm} \odot \\
(fuel\_flow\_ready \odot \\
SPI.spark \odot \text{idle\_spark\_interval} \odot SPI.spark \odot \\
(idle\_combustion\_sensor\_warming\_up\_time \odot \text{Alarm} \odot \\
(CSI.combustion\_going \odot \\
\sigma(\text{furnace:} = \text{WARMING\_UP}) \odot \\
\text{FLI.on} \odot \\
(\text{Controller} \mid \text{Furnace\_warming\_up\_timer}))))))

After opening the water valve the furnace becomes in its normal heating state. During this state the fuel flow is continuously monitored.
7.3. Process control systems

\[ HEATING \triangleq \]
\[ WVI.\text{open} \]
\[ \sigma(\text{water_valve} := \text{OPEN}) \]
\[ \sigma(\text{furnace} := \text{HEATING}) \]
\[ (\text{Controller | Poll\_fuel\_flow\_running}) \]

The shut down routine closes down the fuel valve, idles until this has taken effect, turns out the furnace lamp, waits 5 seconds, stops the motor and disables the furnace for 5 minutes.

\[ SHUTDOWN (\text{mode}) \triangleq \]
\[ FVI.\text{close} \]
\[ (\text{idle}\_fuel\_v\_closing\_time + \delta_{fue} \odot \text{Alarm} \]
\[ \oplus \]
\[ (\text{CSI.combustion\_too\_low,} \times t \odot \]
\[ \text{idle}\_fuel\_v\_closing\_time + \delta_{fue} - t \odot \]
\[ \sigma(\text{furnace} := \text{OFF}) \odot \]
\[ \text{FLI.\text{off}} \odot \]
\[ \text{idle}5 \ast \text{SEC} - \text{fuel\_v\_closing\_time} + \delta_{fue} - t \odot \]
\[ \text{MI.\text{stop\_motor}} \odot \]
\[ \sigma(\text{motor} := \text{STOPPED}) \odot \]
\[ \text{if mode = NORMAL\_SHUT\_DOWN} \]
\[ \text{then (Controller | Disable\_furnace))} \]
\[ \text{else furnace\_stopped} \odot \text{null} \]

The shut down agent can be invoked from two different environments: either from within the normal shut down routine, after which the controller continues its behaviour or from within the Alarm agent, which is specified next.

\[ \text{Alarm} \triangleq \]
\[ SHUTDOWN (\text{ALARM\_MODE}) \mid \text{furnace\_stopped} \odot \text{null} \]

The remaining agents model the various polling and timing actions of the controller.

\[ \text{Disable\_furnace} \triangleq \]
\[ \sigma(\text{furnace} := \text{DISABLED} \odot \]
\[ \text{idle}5 \ast \text{MIN} \odot \]
\[ \sigma(\text{furnace} := \text{OFF}) \odot \]
\[ \text{enable\_furnace} \odot \text{null} \]
Chapter 7. Application

\[ \text{Poll}_\text{fuel_flow_running} \triangleq \]
\[ \text{if } \text{furnace} = \text{OFF} \]
\[ \text{then null} \]
\[ \text{else } \text{idle } \text{FFS}_\text{polling_rate} \circledast \]
\[ \text{FFSI.poll} \circledast \]
\[ \text{FFSI.read_out}(\text{fuel_flow}) \circledast \]
\[ \text{if } \text{abs}(\text{fuel_flow} - \text{fuel_flow_setting}) > \text{fuel_flow_deviation} \]
\[ \text{then Alarm} \]
\[ \text{else } \text{Poll}_\text{fuel_flow_running} \]

\[ \text{Poll}_\text{fuel_flow_start} \triangleq \]
\[ \text{idle } \text{FFS}_\text{start_polling_rate} \circledast \text{FFSI.poll} \circledast \]
\[ \text{FFSI.read_out}(\text{fuel_flow}) \circledast \]
\[ \text{if } \text{abs}(\text{fuel_flow} - \text{fuel_flow_setting}) < \text{fuel_flow_deviation} \]
\[ \text{then } \text{fuel_flow_ready} \circledast \text{null} \]
\[ \text{else } \text{Poll}_\text{fuel_flow_start} \]

\[ \text{Furnace}_\text{warming_up_timer} \triangleq \]
\[ \text{idle } \text{furnace}_\text{warming_up_time} + \delta_{\text{fut}} \circledast \]
\[ \text{if } \text{furnace} = \text{BURNING} \]
\[ \text{then null} \]
\[ \text{else } \text{Timeout}_\text{furnace_warming_up_timer} \circledast \text{null} \]

\[ \text{Motor}_\text{starting_up_timer} \triangleq \]
\[ \text{idle } \text{motor}_\text{starting_up_time} + \delta_{\text{mst}} \circledast \]
\[ \text{if } \text{motor} = \text{RUNNING} \vee \text{motor} = \text{STOPPED} \]
\[ \text{then null} \]
\[ \text{else } \text{Timeout}_\text{motor_starting_up_timer} \circledast \text{null} \]

7.3.4.5 The assembled control component

The assembled control component consists of the composition of the interface agents to both the actuators and sensors and the controlling server itself. All ports shared by the interface agents and the controller are restricted to the control component. The interface of the resulting component consists as usual of the state specification, the ports specification, the various constants and shared subagents. The full specification may be inspected in [211].

7.3.4.6 The whole system

The whole system, including the process part and the control part is now formed by composing the two parts and restricting the composition from
all connections between the interface agents and the actuators and sensors. The only available ports to the experimenter that remain after the restriction are the means to control the main switch. Again the reader is pointed to [211] to study the full specification.

7.3.5 Analysis

The behaviour of the process control system is depending on the presence of certain events in time, e.g. the master switch events and the events generated by the thermometer in the room. As such is the software of the control system constrained by external timing properties. In the case of our process control system the safety and liveness properties not only encompass the termination, divergence and reachability aspects but also timing constraints. In this section these aspects are investigated in some detail.

Both state spaces or traces can be taken as the starting point for the analysis. An abstraction, or simplification of both devices is used to visualise the main characteristics of the semantic models: the continuation chart. A continuation chart is a directed labeled graph. The set of labels is restricted to \( \{\alpha, \iota, \epsilon\} \), where the \( \alpha \) label denotes any external action, \( \iota \) denotes the internal action as usual and \( \epsilon \) denotes any time action. In the graphs the \( \alpha \) labels are pictured as straight arrows, the \( \iota \) labels as dotted arcs and the \( \epsilon \) labels as dashed arcs. Figure 7.16 displays the continuation graphs of all process constituents. The graphs can be derived directly from the agent definitions. All graphs read from left to right and are cyclic, the end nodes coincide with the start nodes. E.g. the continuation chart for the room component shows that the room agent first performs a time action (idling), followed by an external action. Choices between different actions that are possible from a certain state are visualised as multiple arrows starting from the same node. The continuation graphs can be interpreted as abstracted from state transition graphs. Then each arrow denotes a generic transition. When interpreted as abstraction from traces, the graphs are visualisations of regular expressions that generate the trace sets. The graphs can even be interpreted as time charts, where each arrow denote a time progression step. In this interpretation the \( \alpha \) arrows denote waiting time (on the environment to offer a partner synchronization), the \( \iota \) arrows denote zero time progression, as they symbolise internal actions and the \( \epsilon \) arrows symbolize actual time progression by idling. The continuation charts for an abstracted form of the interface devices are presented in figure 7.17. The continuation chart of the controller agent is given in figure 7.18. Here the different paths are labeled with the associated execution state of
Figure 7.16: Continuation chart process actuators and sensors

the agent.

7.3.5.1 Termination properties

When abstracting away the state explosion due to time involvement the state space $SS^F_\epsilon$ of the whole process control system remains finite and small enough to be inspected for deadlocked states or terminal states. The only terminal states in the state space occur after invoking the Alarm agent, that closes down the whole system.

There are no deadlocked states in the state space. But careful inspection of the state space reveals a grave design error. There is a cyclic deadlock due to the design of the SHUTDOWN and Alarm agents. The SHUTDOWN agent activates the Alarm agent in the case of malfunctioning of the fuel valve. Whenever it refuses to close within a precalculated time, registered by the absence of the combustion_too_low action, the Alarm agent takes
over and ..., starts by activating the *SHUTDOWN* agent, which again will activate the *Alarm* agent, etc. The design error was created by miscalculated double use of the series of actions in the *SHUTDOWN* agent. Whenever the fuel valve refuses to close a totally different routine must be invoked to signal the malfunctioning! E.g. an emergency bell must be sounded, to alert bystanders to close the fuel flow by hand.

### 7.3.5.2 Divergence properties

There is no divergence possible within the process control system. A quick inspection of the continuation charts will reveal this fact. After all, divergence is the possibility of endless repeated internal actions. The continuation graphs display that each series of internal action is followed directly by an external or time action, where the length of the series is limited to two.

### 7.3.5.3 Reachability properties

There is a subtle error in the design of the process control system that is revealed by reachability inspection. It addresses the water temperature and home temperature measuring devices. When the heater is activated and the room needs heating, the furnace is started and the water in the furnace is warming up. Assumed that the water temperature in the furnace was beneath the value that signals a too low water temperature, the first crossing of the ctp is just this point, and the *water too cold* signal is generated. The controller then responds by closing down the system by invoking the *Alarm* agent (the furnace is burning!), whereas the expected signal, *water* would be generated much later when the ctp crosses the higher mark on the temperature scale. This fact was revealed first during a test session with the *MOSCA* simulator/rapid prototyper. Inspection of the state space and *MOSCA* specification resulted in locating the trouble spot directly.
Figure 7.18: Continuation chart controller

The same problem may occur when, on the time the system is activated by a master switch event, the room temperature is already less than the temperature on which the room too cold would be generated. The room clearly needs heating, but again the position of the ctp inhibits the generation of the event.

In the case of the water temperature measurement device, the problem is solved by selecting a measurement device that first must be warmed up, just like the water, generates an event when it is on working temperature, and then waits for the temperature to fall, after which it starts all over again.

\[
\text{WATER TEMP SENSOR} \triangleq \\
\text{idle(warming up time())} \odot \text{wout} \odot \\
\text{idle(sensoring())} \odot \text{water too cold} \odot \text{WATER TEMP SENSOR}
\]
In the case of the home temperature measurement device, the extrema should be variable and set by the experimenter. Each time the minimum (or maximum) value is set, the ctp must be consulted to conclude whether a signal should be given.

\[ ROOM_4 \]

\begin{align*}
\text{ports} & \; \text{syn} \; \text{room\_too\_cold} \\
& \quad \text{syn} \; \text{room\_too\_hot} \\
& \quad \text{in} \; \text{set\_low} \quad : \; \text{Temperature} \\
& \quad \text{in} \; \text{set\_thigh} \quad : \; \text{Temperature} \\
\text{state} & \; \text{ctp} \quad : \; \text{Temperature} \\
& \; \text{tlow} \; \quad : \; \text{Temperature} \\
& \; \text{thigh} \; \quad : \; \text{Temperature} \\
\end{align*}

\[ ROOM_4 \triangleq \]

\begin{align*}
\text{(idle}(\text{sensoring}()) \odot & \ (\text{room\_too\_cold} \odot \text{ROOM}_4) \oplus \\
& \ (\text{room\_too\_warm} \odot \text{ROOM}_4) \oplus \\
\text{(set\_low}(\text{nlow}) \odot \sigma(\text{tlow} = \text{nlow}) \odot & \text{if} \; \text{tlow} > \text{ctp} \\
& \quad \text{then} \; \text{room\_too\_cold} \odot \text{ROOM}_4 \\
& \quad \text{else} \; \text{ROOM}_4) \oplus \\
\text{set\_thigh}(\text{nhigh}) \odot \sigma(\text{thigh} = \text{nhigh}) \odot & \text{if} \; \text{ctp} > \text{thigh} \\
& \quad \text{then} \; \text{room\_too\_warm} \odot \text{ROOM}_4 \\
& \quad \text{else} \; \text{ROOM}_4)
\end{align*}

7.3.5.4 Timing constraints

The process control system contains a strict boundary between the two constituents: the set of actuators and sensors, together with the hardware of the process technology and the control system build from both hardware and software components. Over this interface sensor events pass from the process environment to the control system and actuator commands from the control system to the process environment.

Inspired by the work of Dasarathy ([62]) the following constraints can be distinguished. Let \( AC \) denote an actuator command, \( SE \) a sensor event and \( IC \) an internal action \((an, ia)\), where \( an \) denotes an agent or unit name and \( ia \) an internal state changing action.

- \( SE-AC \) constraints. This constraint states that the control system shall respond on a sensor event within an interval of time. The con-
straints are denoted as:

\[ SE-AC[SE\text{-}action, AC\text{-}action, t_m\text{in}, t_m\text{ax}], \]

where \( SE\text{-}action \) and \( AC\text{-}action \) are the names of the involved MOSCA actions, and \( t_m\text{in} \) and \( t_m\text{ax} \) expressions over the domain T. This kind of constraint constrains the behaviour of the control system.

- \( AC\text{-}AC \) constraints. This constraint states that whenever the control system issues an actuator command \( AC_1 \) it is followed by another actuator command \( AC_2 \) within an interval of time, independent of the behaviour of the environment. This kind of constraint is recorded as:

\[ AC\text{-}AC[AC\text{-}action, AC\text{-}action, t_m\text{in}, t_m\text{ax}]. \]

Again this constraint forms a limitation of the behaviour of the control system.

- \( SE\text{-}SE \) constraints, recorded as

\[ SE\text{-}SE[SE\text{-}action, SE\text{-}action, t_m\text{in}, t_m\text{ax}], \]

which specify that the environment should offer two – perhaps related – events within a specific interval of time. This constraint offers information to the control system on the expected behaviour of its environment. If the environment fails to behave accordingly in most of the cases a specific error situation has occurred, and the control system may act precautionary.

- \( AC\text{-}SE \) constraints, recorded as

\[ AC\text{-}SE[AC\text{-}action, SE\text{-}action, t_m\text{in}, t_m\text{ax}], \]

which again specify a constraint on the expected behaviour of the environment.

- \( AC\text{-}IC, IC\text{-}AC, SE\text{-}IC, IC\text{-}SE \) constraints that relate incoming or outgoing actions with internal (state changing) actions, and

- \( IC\text{-}IC \) constraints, that relate two internal (state changing) actions.

Let the following list of constants be defined (and given a specific value).
7.3. Process control systems

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{\text{prmin}} )</td>
<td>the minimum room temperature</td>
</tr>
<tr>
<td>( t_{\text{prmax}} )</td>
<td>the maximum room temperature</td>
</tr>
<tr>
<td>( t_{\text{wumin}} )</td>
<td>the minimum water temperature</td>
</tr>
<tr>
<td>( t_{\text{wumax}} )</td>
<td>the maximum water temperature</td>
</tr>
<tr>
<td>( t_{\text{mst}} )</td>
<td>the motor start-up time</td>
</tr>
<tr>
<td>( \delta_{\text{mst}} )</td>
<td>the motor start-up time margin</td>
</tr>
<tr>
<td>( t_{\text{foo}} )</td>
<td>the fuel valve open time</td>
</tr>
<tr>
<td>( \delta_{\text{foo}} )</td>
<td>the fuel valve open time margin</td>
</tr>
<tr>
<td>( t_{\text{fuc}} )</td>
<td>the fuel valve closing time</td>
</tr>
<tr>
<td>( \delta_{\text{fuc}} )</td>
<td>the fuel valve closing time margin</td>
</tr>
<tr>
<td>( t_{\text{fwu}} )</td>
<td>the furnace warming up time</td>
</tr>
<tr>
<td>( \delta_{\text{fwu}} )</td>
<td>the furnace warming up time margin</td>
</tr>
<tr>
<td>( p_{\text{fs}} )</td>
<td>the fuel flow starting polling rate</td>
</tr>
<tr>
<td>( p_{\text{fr}} )</td>
<td>the fuel flow running polling rate</td>
</tr>
<tr>
<td>( \text{SEC} )</td>
<td>the number of ticks in a second</td>
</tr>
<tr>
<td>( \text{MIN} )</td>
<td>the number of ticks in a minute</td>
</tr>
</tbody>
</table>

The time domain \( T \) is assumed to be discrete. Its dimension is given as ticks.

Given the above constant definitions, the timing constraints for the home heater system can be formulated. Examples of the timing constraints are given in the next list. Let \( \text{MAX} \) denote a not further defined constant denoting some supremum on the time domain.

1. \( AC-AC[FV.\text{close}, M.\text{stop\_motor}, 5 \times \text{SEC}, 5 \times \text{SEC}] \)
2. \( SE-AC[MS.\text{ms\_off\_event}, FL.\text{off}, 0, 5 \times \text{SEC}] \)
3. \( SE-AC[CS.\text{combustion\_too\_low}, FL.\text{off}, 0, 5 \times \text{SEC}] \)
4. \( IC-IC[\sigma(\text{furnace}:= \text{OFF}), \sigma(\text{furnace}:= \text{ON}), 5 \times \text{MIN}, \text{MAX}] \)
5. \( AC-SE[M.\text{start\_motor}, M.\text{motor\_running}, t_{\text{mst}}, t_{\text{mst}} + \delta_{\text{mst}}] \)
6. \( AC-IC[FV.\text{open}, fuel.\text{flow\_ready}, t_{\text{foo}}, t_{\text{foo}} + \delta_{\text{foo}}] \)
7. \( AC-SE[FV.\text{close}, CS.\text{combustion\_too\_low}, t_{\text{fuc}}, t_{\text{fuc}} + \delta_{\text{fuc}}] \)
8. \( AC-SE[FL.\text{on}, WTS.\text{wont}, t_{\text{fwu}}, t_{\text{fwu}} + \delta_{\text{fwu}}] \)
9. \( \ldots \)

Although obviously constrained, the maximum time the water valve may take to open or close is not expressible in terms of timing constraints. The actual opening and closing are within the current specification unobservable actions for the control system. The furnace lamp causes the same problems.
Chapter 7. Application

It is not observable from the context of the control system whether the lamp functions at all.

The timing constraints form a series of proof obligations that must be discharged in a rigorous style\(^5\) to guarantee that the specification matches the requirements. As an example of an outline of such proof the second timing constraint is taken.

**Proposition 7.1** The second timing constraint,

\[ SE-AC[MS.ms\_off\_event, FL.off, 0, 5 \times \text{SEC}] \]

holds for the current heater system specification.

**Proof:** Consider the continuation chart for the Controller agent. The path under inspection is labeled `SHUTDOWN`. It is assumed that the system is working correctly, i.e. the dreaded fuel valve behaves correctly. Now it must be shown that the following series of actions of the controller take less than 5 \times \text{SEC} time units:

\[
\sigma(master\_switch := \text{OFF}) \odot \\
FVI\_close \odot \\
CSI\_combustion\_too\_low, \ast \odot \\
idle t_{fuc} + \delta_{fuc} - t \odot \\
\sigma(furnace := \text{OFF}) \odot \\
FLI\_off
\]

The desired result is shown without a formal derivation. A complete formal treatment would be lengthy, and full of unnecessary details. The formal semantics of MOSCA enables a full formal treatment, but inspection of the continuation charts of the involved agents supplies enough information to show the circumstances under which the timing constraint will hold.

Given that internal actions are instantaneous, the first action takes no time at all. The second action takes 1 time unit, due to the latch in the interface device. The interface device responds directly, under the assumption that each actuator takes its commands whenever it is waiting for them, irrespective of any malfunctioning. It must be the case that the combustion sensor generates the correct event, again due to the assumption that the system behaves correctly. Then the total idling time amounts to \(t_{fuc} + \delta_{fuc}\), and as such it must hold that

\[ t_{fuc} + \delta_{fuc} + 1 < 5 \times \text{SEC}. \]

The involved constants depend on the technical characteristics of the devices which must be selected so that these fall within the stated limits. \(\Box\)

\(^5\) Or even completely formal.
7.3.6 Discussion

7.3.6.1 Outline of the approach

This case study has highlighted the modelling powers of MOSCA in some detail. It has shown an approach that could be summarized as follows.

1. establish a list of objects that on a certain level of abstraction compose the system under study, by applying some form of object oriented paradigm;

2. divide the set of objects with respect to their functionality into actuators, sensors and autonomous objects;

3. model the behaviour of each object up to a certain level of abstraction;

4. compose the set of objects into a working model of the process system;

5. define a control system interface for each class of objects consisting of devices that handle all incoming and outgoing events;

6. design the control component;

7. compose the overall system;

8. model the timing constraints in terms of the actions over the interface and the actions within the control component;

9. analyse the behaviour of the overall system by inspecting carefully chosen state spaces, traces or abstractions of these semantic tools and simulate / prototype (parts) of the system;

10. re-design or adapt the models according to the results of the analysis;

11. refine the objects by applying steps 3 – 10 on particular sub systems or objects, until desired levels of refinement are reached.

The presentation of the objects in the system can be viewed from a dual point.

- In a prescriptive view they form an abstraction of the object of choice, and thus describe a class of real-world objects that implement the specification. The specification constrains the design or choice of a real-world object.
Chapter 7. Application

- In a descriptive view they form a description of the capabilities of an already existing real-world object. Here the real-time object dictates the contents of the specification.

During the development of the system both specifications may evolve, and can be related formally to investigate whether the descriptive specification matches the prescriptive specification.

The approach taken in this case study is based on the currently widely applied paradigms of object orientedness and (top-down) refinement. Although the approach is still highly experimental, it may be developed further to provide a method for the systematic description of process control systems in MOSCA.

7.3.6.2 Comments on the MOSCA notation

This case study has shown that the current notation contains several drawbacks with respect to the ease of usage in general.

- The specification of the port interface of an agent becomes cumbersome in the context of compositions, restrictions and relabelings. The interface of a general form

\[(P \mid Q)[p_1 \mapsto r_1, \ldots] \setminus \{p_i, q_j, r_k \ldots\}\]

is complicated and should be generated automatically, instead of being specified explicitly.

- In the specification several times a specific realization of the following generic form appears:

\[a \circ (P \mid Q)\]

where

\[P \triangleq \text{do-the-P-actions} \circ \text{ready} \circ \text{null}\]

and

\[Q \triangleq \text{ready} \circ \text{do-the-Q-actions}\]

In fact this construction implements a sequencing of agent actions. Other notations based on the CCS model have introduced syntactical shorthands for this kind of phenomena. In LOTOS e.g.

\[a.(P; Q)\]

realizes the same effect in combination with the special stop action that signals the termination of agents.
The introduction of new operators in the language may cause substantial semantic adaptations but may also have a severe influence on the usage of the language: "What is absent in a language, one must not learn to use!" A small set of orthogonal concepts is often easier to understand than a large set of seemingly unrelated concepts. It is my opinion that the introduction of syntactical shorthands in this form, i.e. new operators with specific meaning, is only justified if the modelling power of the notation is substantially extended.

7.3.6.3 Comments on the case subject

The closing part of this presentation discusses some specific properties of the specification with respect to the degree of reality that is captured in the presentation of the different parts and the complexity of the case subject in general.

In reality the heater system will function by electrical power. E.g. it is activated by placing the electrical plug into the mains or de-activated by pulling the cord from the mains. These two real-world actions are not modelled in the current realization. All objects in the process part are modelled in such way that they become into being after the construction of the process from its constituents (in the composition associated to the PROCESS agent). Further they are active infinitely. The main switch has no effect on the behaviour of the process parts. It only constrains the working of the control component. It would take considerable effort to model the presence or absence of electrical power in the whole system.

There is a slightly deviating view in the general concept of the constituents of the process between the process realization and the control part. The objects have been specified as being infallible. Their behaviour is never ending, whereas the control system anticipates on malfunctioning of the system due to faulty components. It is straightforward to model failing components. E.g. a fuel valve with a deteriorating behaviour could be modelled as

\[
\text{REALISTIC}_\text{-FUEL}_\text{-VALVE}
\begin{align*}
\text{ports} & \text{ syn open} \\
& \text{ syn close} \\
\text{state} & \text{ usage : } N \\
\text{inv} & \text{ usage } \triangleq \text{ usage } = 0
\end{align*}
\]
\[ \text{REALISTIC\_FUEL\_VALVE} \triangleq \]
\[ \sigma(\text{usage} := \text{usage} + 1) \circ \]
\[ \text{idle} (\text{breakdown\_timeout(usage)}) \circ \text{null} \circ \]
\[ (\text{open} \circ \text{idle\_fuel\_v\_opening\_time} \circ \]
\[ \text{close} \circ \text{idle\_fuel\_v\_closing\_time} \circ \]
\[ \text{REALISTIC\_FUEL\_VALVE}) \]

There are several other drawbacks in the current design. As stated before there is no control over the realization of both the furnace lamp and the water valve actions. It is not disastrous if the furnace lamp is defect but a defect water valve may cause the furnace to become overheated and malfunction.

Judged by the grade of complexity inherently present in the topic of choice, the case subject represents only a toy application.

The process control systems that form a real challenge for specification and design techniques are e.g. advanced robot-arm control systems [121], plant control [175] or any other really complex process control system. The applied control techniques are simple basic techniques such as feed backward, open loop and closed loop control. Advanced control techniques form another challenge for the specification technique.
8.1 Introduction

This chapter addresses one single aspect related to the transformation of MOSCA specifications into implementations: the implementation of the CCS basic primitives. A language primitive is proposed that directly supports the symmetrical synchronous communication model of CCS. The primitive is designed to support distributed implementation models without any assumptions about the language paradigms and as such it can serve a broad spectrum of existing languages.

The language primitive is based on a specific protocol, denoted by the P2L3S protocol, that enables synchronous inter process communication between two processes within a collection of processes that are connected by a binary tree connection network. The protocol assumes a distributed address space, symmetrical unawareness of synchronization partners and employs a specific form of multicasting messages through the binary tree interconnection network.

A synchronization consists of a simultaneously experienced moment in time by two processes that, during their actions arrive at equally labelled synchronization points. Synchronizations are pure symmetrical and anonymous. The two involved processes are not aware of each others identity. When one process arrives at a synchronization point it waits until some other process arrives at an equally labelled synchronization point. When this fact is established, the two processes continue their actions.

In figure 8.1 (a) a single synchronization between two processes is established. Horizontally the life line of the processes enfold. Upon reaching a synchronization point (boxed) labelled $s$, the process waits until another process reaches an equally labelled point. During a certain running-in pe-
Figure 8.1: Parallel Synchronizations

period in which the two processes become aware of each others existence (denoted by the dashed line pieces in figure 8.1), the synchronization is established (depicted by little circles connected by dashed lines), after which the processes finalize the synchronization and continue their actions asynchronously. In figure 8.1 (b) two parallel synchronizations are established between different processes. It is not possible to establish multi-process synchronizations: each synchronization is point to point.

Synchronizations may become parameterized on data values, thus introducing data-communication. When two processes communicate through a synchronization, there will be one sending process and one receiving process. Sending processes synchronize on output actions and receiving processes synchronize on input actions.

The protocol studied in this paper is developed to serve as inter-process communication paradigm for implementations of languages that support the CCS primitives in a distributed environment. The terms that are used within the context of the protocol are defined in section 8.2. The basic protocol, consisting of a number of tree walks is presented in section 8.3. In section 8.4 several case studies are worked out that explore the details of the P2I3S protocol. Section 8.5 proves some key properties of the protocol. In section 8.6 extensions to the basic protocol are presented that enable e.g.
data communication between processes in synchronization. The chapter is closed with a short section on related work and some conclusions.

8.2 Terminology

There is a set of processes that proceed in parallel with their execution. The processes are connected through a binary tree connection network. The tree consists on internal nodes and leaves (see figure 8.2). All processes occur as leaves in a binary tree connection network. The tree represents always a full binary tree: each internal node has two children, either other internal nodes or leaves.

The P2L3S protocol uses pebbles (P) to mark a connecting path between processes in synchronization consisting of two legs (2L) in three full sweeps (3S) between the involved parties.

- A leave in the tree represents an executing process, that during its lifetime may want to synchronize with another leave. The wish to synchronize is signalled by placing a request at the leave node by the underlying process. In figure 8.2 the binary tree connection network exists above the dotted line. The processes are represented by the ovals under the leave nodes. The small circles represent internal nodes, the boxes leave nodes.

- An active leave can either be first party or second party in a synchronization. A first party is the active leave that signals its arrival and waits for a partner to synchronize. A second party is an active leave that signals its arrival and finds another active leave waiting.

- Synchronizations occur between two leaves in the tree (a tandem). Synchronizations in different tandems can occur simultaneously.

- Each internal node in the tree represents an active component dedicated to relay messages from parent node to child nodes and vice versa. The arriving messages are handled in order of arrival.

- A pebbled path from an active leave up to the root of the tree is called a spine.

- The connection of two spines is called a pivot node.

- A sweep is a walk over nodes in the tree from one active leave to another active leave consisting of two legs. The first leg starts with walking upwards along a spine to the pivot. The second leg continues with walking a spine downwards towards the other leave.
A branch is a connection between an active leave and a pivot consisting of nodes with filled sync holes. The trajec from the pivot to the first party is denoted the first branch of the synchronization trajec. The trajec from the pivot to the second party is denoted the second branch of the synchronization trajec. Two completed branches establish a synchronization.

![Binary Connection Network Tree](image)

Figure 8.2: Binary Connection Network Tree

Internal nodes in the tree are represented by a set of action structures. Each possible label that can occur in the synchronizations of the leaves of the subtree rooted by the internal node are represented within the set.

Each action structure consists of three pebble holes:

- the top hole called sync hole is used to mark the node to be on a trajec from one party to another party.

- the left hole called the left pebble hole is used to mark the node to be in a spine that continues downwards by following the left operand of the node.

- the right hole referenced by the right pebble hole is used to mark the node to be in a spine that continues downwards by following the right operand of the node.

An action structure can hold one of the following seven different values. Each value has a special connotation within the protocol.

- a free node, marks a node that is not currently involved in any phase of a synchronization attempt.
8.2. Terminology

- a left descending spine node within a spine.
- a right descending spine node within a spine.
- a pivot node.
- a claimed node with a left continuation within the branch.
- a claimed node with a right continuation within the branch.
- a false pivot node. This is a remnant of a failed synchronization attempt. It will be turned into a spine node after cleaning up.

The action structure of a leave node consists of two pebble holes, an upper hole, equal to the sync hole, and a lower hole, representing a request to synchronize. A leave with a filled request hole is called an active leave.

The arrival of a leave process waiting on a synchronization point starts a tree walking process consisting of several phases that either result in multicasting the arrival of a process at a synchronization point, or in the realization of a tandem.

The description of the protocol is in terms of:

- events, which in each case denote the arrival of a certain message,
- actions, taken upon the arrival of a certain message,
- action structure values that hold before the event,
- new action structure values resulting from the arrival of the event and
- actions that are taken upon the arrival of a certain event.

The protocol describes a tree walking procedure that is activated on the arrival of each synchronization request. This procedure consists of a number of phases, each defining a series of actions on the nodes in the tree.

A phase is denoted by the number of leg and pass it makes. E.g. phase 2L3S describes the actions on the second leg in the third pass. Each phase is characterised by an message/action block that describes the actions taken in an internal node on the arrival of the characteristic message of the phase. Within the message/action block first the characteristic message (including the direction of arrival: either from the parent or one of the children) and the phase are stated. At the top right of the block a section presents action structure values that form a precondition to all actions in the phase stating that the old state value may not be equal to any of the specified values. The remaining lines describe triples (os, ns, act) with os the old state value
Figure 8.3: Action structure value state changes

before the message, ns the state value after the message and act, the action triggered by the message.

The phases are described by the effect of a message on the state of the action structure within the internal node. The possible changes between the different action structure values are depicted in figure 8.3. Before presenting the detailed description of the protocol a simple example is worked out to get a general idea of the mechanics of the protocol.

Example 8.1 (first scenario: simple synchronization) A simple synchronization between two parties involves the sequence of steps depicted in figure 8.4. The first party to arrive (a) starts to pebble its spine to the top node (depicted as a heavy solid black line) and becomes passive afterwards, waiting on another party. This phase acts as a multicast of the message that announces the arrival of the first party. Starting in node 1 in the Phases State Diagram (figure 8.5), the protocol proceeds to state 2. A second party arrives (b) and starts pebbling its spine. When it encounters an spine node, i.e. a node in the spine of another party it becomes a second party. Again starting in state 1 in figure 8.5 the protocol proceeds to state 3. The tree walk continues in the direction of the first party (c), claiming the spine (depicted as replacing the solid black with the dark gray line). When the first party is reached, the protocol reaches state 4 and continues backwards to the pivot reaching state 5 and then continues by claiming the
Figure 8.4: Simple synchronization between two parties
spine from the pivot to the second party. If this action is ended successfully
the synchronization is a fact: the whole traject from the second party to
the first party is claimed (e) and the protocol reaches state 6. Now cleaning
up closes the synchronization, in which two processes act in parallel:
(i) all pebbles from the second party to the first are picked up (f) and (ii)
the pebbles from the first party from the pivot to the top node are picked
up. The whole tree is in its original situation (state 8). The actions of the
involved parties can be summarized in terms of state successions executed
by the procedures that were started at their arrival. Let \( P(l) \) denote such
a succession of states, then we get:

\[
\begin{align*}
P(1): & \quad 1 \rightarrow 2 \\
P(2): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8
\end{align*}
\]

\[\square\]

8.3 The P2L3S Protocol

The phases in the P2L3S protocol occur after each other according to the
state diagram of figure 8.5.

Figure 8.5: P2L3S Phases State Diagram

A complete synchronization consists of the following (message, phase) tu-
uples:

- (pebble ↑, PS) : pebbles a spine,
- (pebble ↑, 1L1S) : pebbles a spine of a second party until reaching a
  pivot.
- (claim1 ↓, 2L1S) : claims the first branch of the synchronization,
8.3. The P2L3S Protocol

- \((\text{claim}1 \uparrow, 1L2S)\) : after a successful 2L1S phase the leg from the first party to the pivot is travelled backwards to start the next phase,
- \((\text{claim}2 \downarrow, 2L2S)\) : claims the second branch of the synchronization,
- \((\text{cleanup} \uparrow, 1L3S)\) : cleans the second branch of the synchronization, and
- \((\text{cleanup} \downarrow, 2L3S)\) : cleans the first branch of the synchronization,
- \((\text{unpebble} \uparrow, CS)\) : cleans the spine upwards towards the root.

Two failure situations can occur (The ‘R’ suffix in the phase name suggest a reverse direction of movement):

- \((\text{unclaim}1 \uparrow, 2L1SR)\) : unclaims the first branch after an unsuccessful 2L1S phase,

and

- \((\text{unclaim}2 \uparrow, 2L2SR)\) : unclaims the second branch after a failure in the 2L2S phase, followed by
- \((\text{unclaim}1 \downarrow, 1L2SR)\) : that travels down the first branch towards the first party, followed by
- \((\text{unclaim}1 \uparrow, 2L1SR)\) : that unclaims the first branch.

In the detailed description of the phases the action blocks present only the tree walking actions between internal nodes. The actions stop whenever a top node or leave is reached.

**PS** — On arrival of a first party for a synchronization the spine is pebbled from the leave to the top node. Each node on the spine gets a pebble. Pebbling occurs either in the left or in the right pebble hole. The specific choice marks the flow of the spine downwards. A left pebble points to the left operand as the next node in the spine, a right pebble to the right operand.

When during pebbling a claimed node is reached, the pebbling process is postponed to the moment the node is free of any claims. During the waiting period the node is said to be disabled. It becomes enabled again when freed and pebbling continues.

When during pebbling a true or false pivot is reached, the pebbling process is also postponed until the nodes are cleared. A pivot can be encountered as remnant of a failed synchronization attempt. Example 8.6 will present such a case.
### Event: pebble \[\uparrow\]

<table>
<thead>
<tr>
<th>Phase: PS, 1L1S</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![Node A]</td>
<td>![Node B] V</td>
</tr>
<tr>
<td>![Node C] V ![Node D]</td>
<td>![Node E]</td>
</tr>
<tr>
<td>![Node F], ![Node G], ![Node H], ![Node I]</td>
<td>![Node J], ![Node K], ![Node L], ![Node M]</td>
</tr>
</tbody>
</table>

### 1L1S — On arrival of another party the protocol proceeds, without assuming to be second party in state 1, by pebbling the spine. Then during pebbling it is established that the current party was not the first. Whenever during pebbling the spine a node is encountered that is pebbled we know there is (potentially) another candidate waiting (the first party) on the leave of the spine. Several cases can occur:

(a) The operator node carries the first party: in this case the partner is found directly.

(b) The first party is a direct child of one of the operator nodes on the spine of the second party: pebbling stops: a first party is located.

(c) The first party is a sibling of one of the operator nodes on the spine of the second party. If we find that whole spine downwards intact a synchronization may be realized. Instead of going up with pebbling the protocol is going down to look for the leave, filling the sync hole. Each filled sync hole marks the capture of a node on the spine. Once captured the node must be released first before it can be used in another synchronization effort.

### 2L1S — If, during claiming the first branch going downwards, we come on a pivot of another candidate, or an empty node, this try for synchronization failed. The first branch is unclaimed.

When a false pivot is reached, claiming will be postponed until the node is turned into a spine node.
8.3. The P2L3S Protocol

<table>
<thead>
<tr>
<th>Event: \textit{claim1} ↓</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 2L1S</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![State Diagram]</td>
<td>![State Diagram]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event: \textit{unclaim1} ↑</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim1 ↓</td>
</tr>
<tr>
<td>wait( )</td>
</tr>
</tbody>
</table>

\textbf{1L2S} — After locating the partner, and securing the path from the pivot to the first party by claiming the operator nodes with sync pebbles, the path from the second party to the pivot should be claimed also. This is done in a second sweep, now in opposite direction, from the first party to the second party. The \textit{claim} action first travels up to the pivot. If the pivot holds a postponed pebbler, the pebbler is terminated (again see example 8.6).

<table>
<thead>
<tr>
<th>Event: \textit{claim1} ↑</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 1L2S</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![State Diagram]</td>
<td>![State Diagram]</td>
</tr>
</tbody>
</table>

\textbf{claim1} ↑

\textbf{2L2S} — It then descends the spine towards the second party. If the second party gets reached the synchronization is a fact. If during the claim process another pivot, an occupied node or an empty node is reached the synchronization effort must be abandoned: cleaning up follows (phases 2L2SR, 1L2SR and 2L1SR).

<table>
<thead>
<tr>
<th>Event: \textit{claim2} ↓</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 2L2S</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![State Diagram]</td>
<td>![State Diagram]</td>
</tr>
</tbody>
</table>

\textbf{unclaim2} ↑

\textbf{claim2} ↓
**1L3S** — Both branches of the trajec from second party to first party are secured: the synchronization is a fact. The protocol enters the third sweep of the synchronization. First the leave of the operator node that acted as second party is informed of the end of the synchronization. Again the complete path from the second to the first party is visited, meanwhile taking up all pebbles used to mark the path. All disabled nodes on the path become enabled again.

<table>
<thead>
<tr>
<th>Event: cleanup ↑</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 1L3S</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![symbol1]</td>
<td>![symbol2]</td>
</tr>
<tr>
<td>![symbol3]</td>
<td>![symbol4]</td>
</tr>
<tr>
<td>![symbol5]</td>
<td>![symbol6]</td>
</tr>
</tbody>
</table>

**2L3S** — On arriving at the first party its leave is signalled the end of the synchronization.

<table>
<thead>
<tr>
<th>Event: cleanup ↓</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 2L3S</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td>![symbol7]</td>
<td>![symbol8]</td>
</tr>
</tbody>
</table>

**CS** — Finally the spine from the pivot upwards is cleared. If during the rising towards the top node a pivot is encountered that links another spine cleaning up stops. If a pebbled node is encountered with an empty hole in the direction of arrival (a remnant of an earlier pivot) cleaning the spine stops in order not to destroy the spine of an other leave.
8.3. The P2L3S Protocol

The two failure situations are described next.

2L1SR. — This phase is started whenever during the first sweep downwards a pivot or an empty node is encountered.

- A pivot can be encountered if another party, a third party further down the spine towards the first party, enters and starts claiming to the first party before the second party.
- An empty node can be encountered as a result of a successful synchronization, involving the first party and a third party, in the subtree rooted by the empty node. The cleaning up after the synchronization has emptied the spine from the first party upwards.

From the current node upwards to the pivot all pebbles resulting from claiming are removed. Coming at the pivot the protocol enters state 9 and starts to repebble the spine towards the top node. Repebbling is necessary due to possible bounced claim1 actions.

2L2SR. — If during the second sweep downwards another pivot is reached, or an empty node is encountered the synchronization between the second and the first party fails, the first party remains and the situation on the first branch must be restored. The actions are the following: first the branch from the current node to the pivot is cleaned-up. All
pebbles laid down to claim are removed. Then the pivot is turned into a false pivot, and the next phase is started.

<table>
<thead>
<tr>
<th>Event: unclaim2 ↑</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 2L2SR</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>

1L2SR — the first branch is travelled downwards to the first party. Upon reaching the first party, phase 2L1SR follows to clear the first branch.

<table>
<thead>
<tr>
<th>Event: unclaim1 ↓</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: 1L2SR</td>
<td></td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td><img src="image4" alt="Diagram" /></td>
<td><img src="image5" alt="Diagram" /></td>
</tr>
</tbody>
</table>

R1L1S — This phase establishes a repelbling walk towards the top node. The pivot of a synchronization attempt is in fact a true pivot only when both branches are secured. During the interval between the formation of the pivot and the moment both branches are a fact the pivot is in fact untrue. When during this interval a claim attempt reaches the pivot, this attempt bounces back to its originating pivot and transforms into a spine pebbling activity towards the top node. If during the interval another synchronization takes place within the subtree of the untrue pivot including on of the leaves that formed the untrue pivot, the bounced claim will not try again and the remaining party becomes obsolete unless the pebbling activity upwards is performed again.
8.4 *P2l3S scenarios*

<table>
<thead>
<tr>
<th>Event: <code>repebble</code> ↑</th>
<th>Exceptions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase: RS, R1L1S</td>
<td>Ø</td>
</tr>
<tr>
<td>old state</td>
<td>new state</td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Just as in CS and 1L1S repebbling turns to claiming when a pebbled node with a pebble at the opposite hole is encountered. Repebbling stops if a claimed or pivot node is encountered.

### 8.4 P2l3S scenarios

To appreciate the mechanics of the protocol several cases are presented that capture the different situations that are plausible. Next the protocol is analysed more formally.

**Example 8.2** *(second scenario, in which three parties are involved, and the synchronization takes place between party 2 and party 3.)* In figure 8.6 we find three parties involved in establishing a synchronization.

In figure (a) party 1 arrives and pebbles its spine. Then in (b) party 2 arrives, starts pebbling its spine and comes to spine node p1. So instead of continuing pebbling to the top, it starts in (c) claiming the first branch. But then, in (d) party 3 arrives and starts pebbling towards the top node. In p2 it changes direction (e) and claims the spine towards leave 2. Meanwhile the spine towards leave 1 is completely claimed and the process continues with starting to claim the spine from p1 towards leave 2. In (f) synchronization between leave 3 and leave 2 becomes a fact. The process has finished its round trip starting at leave 3, travelling over p2 towards leave 2, claiming the spine from p2 to leave 2, and back again to leave 3, claiming the spine from p1 towards leave 3. The process that originated from leave 2 stops coming back to p2 and finds it to be a pivot. It cannot proceed any further and the back sweep from leave 1 to leave 2 is not completed. In (g) the last sweep between leave 3 and leave 2 is depicted: cleaning up. The other process has to clean up also (h), restoring the original situation on the spine.
Figure 8.6: Three parties: a 2 — 3 synchronization
from party 1 to the pivot p1. Again the actions of the involved parties are summarized in terms of phase (state) successions as follows:

\[
\begin{align*}
P(1): & \quad 1 \rightarrow 2 \\
P(1): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 11 \rightarrow 12 \rightarrow 9 \\
P(1): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8
\end{align*}
\]

The next example presents a case in which the first branch becomes obsolete.

**Example 8.3** (third scenario, in which three parties are involved, and the synchronization takes place between party 1 and party 3.)

![Diagram](image)

**Figure 8.7:** Three parties: a 1 \(\rightarrow\) 3 synchronization

In figure 8.7 the protocol proceeds by pebbling the spine of party 1 (a). In figure (b) the second party arrives and proceeds by pebbling. Again on
arriving at node p1 it becomes a pivot and the first branch between party 2 and party 1 is on its way to become secured (c). In the meantime party 3 is arrived and coming upon node p2 turns it into a pivot and continues in the direction of party 1 securing the first branch between party 3 and party 1. Thus when the party 2 attempt at securing the first branch encounters p2, it stops, the attempt is abandoned and the path from p1 to p2 cleaned up (f). The state successions now become:

\[
\begin{align*}
P(1): & \quad 1 \rightarrow 2 \\
P(2): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 9 \\
P(3): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8
\end{align*}
\]

\[\square\]

**Example 8.4 (Delayed pebbling)** In figure 8.8

![Diagram](attachment:image.png)

\[\text{Figure 8.8: Delayed pebbling}\]

S1 denotes a subtree rooted with node p3. A leave l in S1 arrives to synchronize and pebbles its path to the top node. Next leave marked p arrives (b) and secures the first branch (c). Finally leave marked q arrives, pebbles up to node p2 and finds it claimed. Pebbling is postponed until node p2 is free again. The state successions:

\[
\begin{align*}
P(l): & \quad 1 \rightarrow 2 \\
P(p): & \quad 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \\
P(q): & \quad 1
\end{align*}
\]

The actions may proceed in several directions. In figure 8.9 two different continuations are sketched.
8.4. P2l3S scenarios

Figure 8.9: Pebbling Resumed

In (d) the synchronization between p and l in S1 has become a fact and after cleaning up p2 is freed. Pebbling from q continues upwards towards the top node. The completed state successions become:

\[
\begin{align*}
P(l) & : 1 \rightarrow 2 \\
P(p) & : 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \\
P(q) & : 1 \rightarrow 2
\end{align*}
\]

In (e) another continuation is sketched. Now m, a rival to p in S1 has taken the synchronization and p remains. The process of p continues by unclaiming the path to p3 in phase 2L1SR and repebbles from p1 towards the top node. The resumed pebbling of q reaches p1 and the normal procedure follows. The state chains become:

\[
\begin{align*}
P(l) & : 1 \rightarrow 2 \\
P(m) & : 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \\
P(p) & : 1 \rightarrow 3 \rightarrow 4 \rightarrow 9 \rightarrow 10 \\
P(q) & : 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8
\end{align*}
\]

The next example presents a scenario in which repebbling is needed.

**Example 8.5 (Repebbling)** In figure 8.10 a scenario develops that includes repebbling. Somewhere in the tree S1 a first party arrives and pebbles to the top (fig (a)). Then q arrives and creates the pivot p2 and starts claiming towards the first party in S1 (fig (b)). Next party p arrives creates
pivot p1 and starts claiming towards the leave on the spine. Upon arriving at p2 the claiming process bounces against pivot p2 and phase 2L1SR unclaims the path from p2 to p1. In the meantime within subtree S1 a rival to q arrived and succeeded in forming a complete synchronization.

Situation (d) of figure 8.11 arrives: q has to unclaim the second and first branches and p has unclaimed the path from p2 to p1 and repebbles to the top node. This action is not essential in this situation as the spine above p1 has not been altered and still functions as multicast message to all processes that are descendant leaves of the spine above p1. Yet in the case of process q it becomes essential. In figure 8.11 (e) the path between p3 and p2 has been cleared by the cleanup process of the synchronization that took place in the subtree S1. The pivot p2 has turned from false pivot (resulting from the unclaim2 action of q) into a spine node. Suppose that repebbling for q starting from p2 does not occur. If no other processes arrive the situation in the tree holds two processes, p and q that could synchronize but never will! Since p is waiting, its process in state 2 of the protocol and q is also waiting, nothing will happen. Now repebbling of q after unclaiming the first branch from q to p2 will assure that waiting processes like p will be found.

\[
\begin{array}{c}
P(l): \quad 1 - 2 \\
P(m): \quad 1 - 3 - 4 - 5 - 6 - 7 - 8 \\
P(p): \quad 1 - 3 - 4 - 9 - 10 \\
P(q): \quad 1 - 3 - 4 - 5 - 6 - 11 - 12 - 9 - 3 - 4 - 5 - 6 - 7 - 8
\end{array}
\]
Figure 8.11: Repebbling 2

The last example presents a case that presents a postponed pebbler on a pivot node.

**Example 8.6 (postponed pebbler on pivot)** The scenario starts with the arrival of a first party (see figure 8.12). Next a second party arrives and starts pebbling towards p1 (figure 8.12 (a)). It is assumed that the 2L1S phase of party 2 takes a while, and during the claiming towards party 1 party 3 arrives and takes party 2 as partner (figure 8.12 (b)) in a synchronization. Cleaning up afterwards proceeds by cleaning the spine in phase CS from pivot p3 upwards towards node t. Upon reaching p1 the unpebbling stops and lets p1 intact. Now p4 arrives and starts pebbling towards p1 (figure 8.12 (c)). Being still a pivot, the pebbling stops and is postponed. Now p1 holds a postponed pebbler that continues as soon as p1 is freed. Although, when finally the 1L2S phase of the procedure that originated by party 2 reaches p1, it discovers the postponed pebbler, and continues with 2L2S in the direction of party 4, after having terminated the postponed pebbler on p2.

| $P(1)$ | 1 — 2 |
| $P(2)$ | 1 — 3 — 4 — 5 — 6 — 7 — 8 |
| $P(3)$ | 1 — 3 — 4 — 5 — 6 — 7 — 8 |
| $P(4)$ | 1 — |
8.5 P2l3S analysis

Definition 8.1 Let $I$ be the domain of internal nodes of a binary tree and $L$ the domain of all leaf nodes. Let $N$ be $I \cup L$. Let $n \in N$ then

- $T(n)$ denotes the tree with top node $n$,
- $C_l(n)$ is the left child of $n$ and $C_r(n)$ is the right child of $n$, if $n \in I$, $C_l(l)$ and $C_r(l)$ for $l \in L$ are undefined.
- $L(n)$ denotes the set of leaves of the tree with root $n$. If $n \in L$ then $L(n) = n$.
- $S(n)$ is the shortest path from the node $n$ to the top node of the tree. This path is called the spine of $n$ and is represented by the
set of nodes on the path, thus the signature of $S$ is $\mathit{N} \rightarrow \mathit{N}$-set. A fragment of a spine is denoted by $[l, u]$ which denotes that part of a spine starting with and including node $l$ and rising towards and including $u$, or by $(l, u)$ which denotes the part of the spine between $l$ and $u$ or by a mixed form like $[l, u)$ or $(l, u]$. The number of nodes in a spine $[l, u]$ is denoted by $l([l, u])$. Given a spine $s$ the predicate is-pebbled$(s)$ checks whether all nodes in $s$ are pebbled.

A pivot resulting from the meeting point of two spines $S(x)$ and $S(y)$ is denoted as $p_{x,y}$ where $S(x)$ appeared before $S(y)$ and thus $p$ came into existence when $S(y)$ met $S(x)$.

- $D(n)$ denotes the depth of the tree with root $n$. The depth is defined as the length of the longest spine in $T(n)$.
- $N(n)$ denotes the nodes in $T(n)$.

For each of the action structure values there is a predicate that tests if its argument holds the value denoted by the predicate. These predicates appear e.g. as is-∗$(n)$.

As the action of different synchronization attempting processes may occur in parallel, a notion is needed that expresses the time subsequent actions will take. The next definition relates actions with time durations.

**Definition 8.2** (atu's) An action time unit (or short atu) is the unit time duration of any action in the tree.

In the following analysis we take the notion time equivalent to a number of performed atu's.

**Definition 8.3** (full synchronization) A full synchronization comprises all actions that occur after the creation of the pivot until the pivot is cleared and consists of the phases 2L1S, 1L2S, 2L2S and 1L3S.

**Definition 8.4** (completed synchronization) A completed synchronization comprises all actions that occur after the creation of the pivot until the end of the protocol and consists of the phases 2L1S, 1L2S, 2L2S and 1L3S, 2L3S and CS.

First the termination of the protocol is analysed. Consider again figure 8.5. To prove the termination of the protocol it must be shown that each procedure started at the arrival of an active leave terminates. These procedures consist of the phases depicted in figure 8.5. So it must be showed that (a) each phase terminates and (b) the cycles in figure 8.5 are executed
at most a finite number of times. It is assumed that the phase succession
depicted in figure 8.5 is working correctly.

Each phase consists of a finite number of steps, limited by the depth
of the tree. There are two different kind of phases: with or without wait
states. Phases without wait states run in $O(d)$ atu’s. For the phases that
contain wait states it must be shown that the waiting time is bounded.
Two different cases occur: pebbling and repebbling (phases PS and 1L1S,
R1L1S and RS) and claiming (phase 2L1S).

The pebbling and repebbling phases contain wait states in which the
waiting depends on claimed nodes. The maximum number of atu’s a node
can be claimed depends on the maximum time it takes to complete a syn-
chronization attempt. Therefore the maximum time it takes to complete a
full synchronization is analysed first. A full synchronization under a pivot
$p$ with $D(p) = d$ takes:

21S : d atu’s
1)2S : d atu’s + all atu’s it takes on waiting on false pivots
212S : d atu’s
1)3S : d atu’s

Let $Sync(x, y)$ denote a synchronization attempt between the nodes $x$ and
$y$, with $x$ first party and $y$ second party. Further let $T_{FS(n)}$ denote the
number of atu’s of a full synchronization (FS) in a tree with depth $n$. The
analysis assumes full binary trees. A tree with depth 0 consisting of only a
single leave does not enables any synchronization, so $T_{FS(0)} = 0$ (see figure
8.13). A tree with depth 1 can perform exactly one full synchronization
at a time between party 1 and 2, taking 4 atu’s, so $T_{FS(1)} = 4$. Figure
8.13 (c) shows a tree with depth 3. In $Sync(1, 2)$ phase 2L1S runs toward
1 and visits node $a$. This node is either pebbled and then $T_{FS(2)}$ takes 8
atu’s, or is involved in $Sync(1, p)$ and $Sync(1, 2)$ fails. So here $Sync(1, 2)$
has no wait states. Let’s examine figure (d). Here $Sync(1, 2)$ runs to node
b, which can be a false pivot due to a failed $Sync(1, p)$ attempt that failed
on a succeeded $Sync(r, p)$. Now there is a wait state. Waiting will take
maximally as long as $Sync(r, p)$ runs plus 1 action to free node $b$.

Let $T_{W(d)}$ denote the waiting time during 2L1S on a spine node on
depth $d$ under the pivot. The emerging pattern shows that

\[
T_{FS(0)} = T_{W(0)} + 4.0 \\
T_{FS(1)} = T_{W(1)} + T_{W(0)} + 4.1 \\
T_{FS(2)} = T_{W(2)} + T_{W(1)} + T_{W(0)} + 4.2 \\
\ldots
\]
Figure 8.13: Trees with depth 0 to 3

and

\[ T_{W(n)} = T_{FS(n-1)} + 1 \text{ for } n > 0 \]  

(8.1)

So we can deduce that \( T_{FS(n)} \) will have the following value:

\[
T_{FS(n)} = \begin{cases} 
4 & \text{if } n = 1 \\
8 & \text{if } n = 2 \\
4n + \sum_{i=1}^{n-2} (T_{FS(i)} + 1) & \text{otherwise}
\end{cases}
\]  

(8.2)

To ease the analysis of the number of atu’s denoted by (8.2) we will use as an upper bound for \( T_{FS(n)} \)

\[
T_{FS(n)} < \begin{cases} 
5n & \text{if } n \leq 2 \\
5n + \sum_{i=1}^{n-2} T_{FS(i)} & \text{otherwise}
\end{cases}
\]  

(8.3)

which follows easily from (8.2). Let’s expand some terms to see how the value of \( T_{FS(n)} \) develops.

\[
T_{FS(n)} \quad < \quad 5n + T_{FS(n-2)} + T_{FS(n-3)} + \ldots \\
< \quad 5n + 5(n - 2) + T_{FS(n-3)} + 2T_{FS(n-4)} + \ldots \\
< \quad 5n + 5(n - 2) + 5(n - 3) + 2T_{FS(n-4)} + 3T_{FS(n-5)} + \ldots
\]
which can be written in a closed formula as

\[ T_{FS(n)} \begin{cases} 
5n & \text{if } n \leq 2 \\
5n + \sum_{i=1}^{n-2} F_{n-i-1} \cdot i & \text{otherwise}
\end{cases} \]  

(8.4)

where \( F_i \) denotes the \( i \)th Fibonacci number. Although this number grows rapidly (exponential with \( \Phi = \frac{1+\sqrt{5}}{2} \) as base) it is less than the order of nodes in the tree, which is \( O(2^n) \). The right hand side of inequation 8.4 can be rewritten, giving

\[ T_{FS(n)} \begin{cases} 
5n & \text{if } n \leq 2 \\
5n + \frac{1}{\sqrt{5}}((n-1)\sum_{i=1}^{n-2} \Phi^i - \sum_{i=1}^{n-2} i \cdot \Phi^i) & \text{otherwise}
\end{cases} \]  

(8.5)

by substitution of \( F_i \) with \( \frac{\Phi^n}{\sqrt{5}} \). The number of nodes in the tree grows with a factor 2. The number of \( T_{FS(n)} \) grows approximately with a factor \( \Phi \approx 1.61803 \). To enforce this claim, I will show that the function \( t(n) \) defined as

\[ t(n) = \begin{cases} 
5n & \text{if } n \leq 2 \\
5n + \sum_{i=1}^{n-2} t(i) & \text{otherwise}
\end{cases} \]  

(8.6)

can be written in the closed form

\[ t(n) = 5(F_{n+2} - 1) \]  

(8.7)

which is clearly approximately equal to \( 5 \Phi^{n+2} \) for \( n \) large enough.

Starting from (8.6) follows the extended equality

\[ 2t(k) + t(k - 1) = 5k + \sum_{i=0}^{k} t(i) \]  

(8.8)

which will form the basis of the derivation. To find a close form for functions defined using a summation over function values like (8.6) generating functions are in general a welcome help. Let

\[ G_t(x) = \sum_{n} t(n) x^n \]  

(8.9)

where if not stated otherwise \( n \) ranges from 0 to infinity. The derivation of formula (8.7) that assigns values for \( t(n) \) runs in four steps:
8.5. **P2l3S analysis**

1. from (8.8) an equation can be derived in terms of power series, which subsequently

2. can be expressed in terms of $G_t$, and $x$ and then

3. from this equation follows a closed formula for $G_t$ as function of $x$ which

4. is transformed back into a power series, that finally results in matching values for the $t(n)$.

The first step delivers:

$$
\sum_n (2t(n) + t(n-1))x^n = \sum_n (5n + \sum_{i=0}^k t(i))x^n \quad (8.10)
$$

It is easy to derive that

$$
\frac{5x}{(1-x)^2} = \sum_n 5nx^n \quad (8.11)
$$

To translate equation (8.10) to a form in terms of $G_t$ a few simple manipulations of the basic generating function (8.9) are needed.

$$
\frac{1}{1-x} G_t(x) = \sum_n \sum_k t(k)x^k = \sum_{m=0}^\infty (\sum_{i=0}^m t(i))x^m \quad (8.12)
$$

and

$$
x G_t(x) = \sum_n t(n)x^{n+1} = \sum_{n=1}^\infty t(n-1)x^n \quad (8.13)
$$

Now step 2 is concluded by substituting (8.11) (8.12) and (8.13) in (8.10):

$$
2G_t(x) + xG_t(x) = \frac{5x}{(1-x)^2} + \frac{G_t(x)}{1-x} \quad (8.14)
$$

which after re-arranging (step 3) results in

$$
G_t(x) = \frac{5}{(1-x)(1-x-x^2)} \quad (8.15)
$$

The two factors in the denominator expand to:

$$
\frac{1}{1-x} = \sum_n x^n \quad (8.16)
$$

$$
\frac{1}{1-x-x^2} = \sum_n F_n x^n \quad (8.17)
$$
where $F_n$ represent again the fibonacci numbers. These two results can be used after splitting the fraction in (8.15) which leads to

$$\frac{5}{(1-x)(1-x-x^2)} = 5\left(\frac{x+2}{1-x-x^2} - \frac{1}{1-x}\right)$$

(8.18)

The first term from this result can be rewritten to

$$\frac{1}{1-x-x^2} - \frac{1-x}{x^2}$$

(8.19)

which expands to $\sum F_{n+2}x^n$. Thus the whole fraction (8.15) expands to

$$G_t(x) = 5\left(\frac{x+2}{1-x-x^2} - \frac{1}{1-x}\right)$$

(8.20)

$$= 5\left(\sum F_{n+2}x^n - \sum x^n\right)$$

(8.21)

$$= \sum 5(F_{n+2} - 1)x^n$$

(8.22)

$$= \sum t(n)x^n$$

(8.23)

from which follows the closed form for the values for $t(n)$. The values for $t(n)$ equal $T_{FS(n)}$ so finally the result becomes:

$$T_{FS(n)} = 5(F_{n+2} - 1)$$

(8.24)

Inspection of the values of $T_{FS(n)}$ and $2^n$ shows that the number of processes in a full tree of depth 9 outgrows the value of $T_{FS(n)}$.

The computed values for $T_{FS(n)}$ are worst-case values for synchronizations in a full binary tree connection network. A full tree for $n$ sufficiently large will in practice seldom occur and the now computed worst-case number is far bigger than the worst-case number for $T_{FS(n)}$ over an arbitrary tree.

So far the analysis has shown that the waiting time in 2L1S is bound by

$$\tau_{2L1S(n)} < \sum_{i=1}^{n-2} F_{n-i-1} \cdot t = T_{FS(n)} - 5n \text{ for } n > 2$$

(8.25)

Now the pebbling and repebbling waiting time $T_P(n)$ will be examined. Again consider figure 8.13. Suppose party 1 arrives and starts to pebble

\footnote{see e.g. [86] chap. 7.}
8.5. P2I3S analysis

\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\(n\) & \(T_{FS(n)}\) & \(2^n\) \\
\hline
1 & 5 & 2 \\
2 & 10 & 4 \\
3 & 20 & 8 \\
4 & 35 & 16 \\
5 & 60 & 32 \\
6 & 100 & 64 \\
7 & 165 & 128 \\
8 & 270 & 256 \\
9 & 440 & 512 \\
10 & 715 & 1024 \\
\hline
\end{tabular}
\end{center}

Figure 8.14: Atu's in Synchronizations

its spine. In figure (a) pebbling takes 0 atu's, so \(T_{P(0)} = 0\). In figure (b) pebbling takes 1 atu without any wait states, so \(T_{P(1)} = 1\). Figure (c) shows that if party 1 arrives it may find node a claimed by e.g. \(Sync(p, 2)\), which will take \(T_{FS(2)}\) time and leads to \(T_{P(2)} = 2 + T_{FS(2)}\). Inspecting figure (d) both nodes a and b can be claimed, and their synchronizations will in the worst-case run over the top node of the tree. Continuing the reasoning leads to the following formula for \(T_{P(n)}\):

\[ T_{P(n)} < n + n \cdot T_{FS(n)} \]  \hspace{1cm} (8.26)

If fact we now have proven the following lemma.

Lemma 8.1 Let \(p\) be a pivot and \(D(p) = d\) in a tree \(T(t)\) with \(D(t) = n\).

- waiting time during claiming in phase 1L2S starting from \(p\) is bound by \(O(T_{TS(d-2)})\).
- waiting time during pebbling in phase PS, 1L1S, R1L1S and RS in \(T(t)\) is bound by \(O(n \cdot T_{FS(n)})\).

Next it is shown that cycling through phases 3-4-9 and 3-4-5-6-11-12-9 is bound by the depth of the tree. Consider figure 8.15. \(Sync(1, 2)\) is in progress but 2L1S a pivot resulting from \(Sync(1, r1)\) is encountered. 2L1S fails and the procedure enters state 9. During repelbling from \(p\) to
Figure 8.15: Cycling through states 3–4–9

t each time we find a first party, say \( l \) the procedure continues with 2L1S, re-entering state 4. Suppose we have

\[
\forall n \in \langle p, t \rangle \cdot \exists n' \in L(n) \cdot p \notin L(n) \wedge \text{is-pebbled}([n, rn]) \wedge n' \in \langle n, rn \rangle \cdot \text{is-} \bigodot (n')
\]

that is, the situation resulting from the \( \text{Sync}(1, 2) \) attempt repeats itself for each node on the spine \( \langle p, t \rangle \). Then the cycle 3–4–9 repeats itself for each node in \( \langle p, t \rangle \), thus the maximum number of executed cycles is bound by \( O(l([p, t])) \). A similar reasoning leads to an equal bound on the number of executed cycles 3–4–6–11–12–9 equal to \( O(l([p, t])) \). We have proven the following lemma.

**Lemma 8.2** The number of executed cycles within the P2L3S protocol is bounded by \( O(l([p, t])) \) where \( p \) is the pivot node on which the 1L1S phase turns to phase 2L1S and \( t \) the top node of the tree.

\[\blacksquare\]

The result of this analysis is the proof of the next proposition.

**Proposition 8.1 (Termination of the P2L3S protocol)**

1. Each phase in the P2L3S protocol performs a number of actions proportional to the depth of the tree, with bound waiting time during pebbling and claiming.

2. The number of executed cycles resulting either from failed claiming in 2L1S or 2L2S is bound by the depth of the pivot within the tree created by the 1L1S phase.
8.5. P2I3S analysis

Proof: By lemma 8.1 and 8.2.

The following analysis is set up to prove that there can never be a situation that two active leaves appear within the tree, that are both waiting on a synchronization over shared action label.

First we prove a simple proposition that relates the creation of pivots with the occurrence of completed synchronizations.

Proposition 8.2 (Causality of completed synchronizations) Let \( l_1, l_2 \in L(t) \) and \( p = S(l_1) \cap S(l_2) \). Let \( l_1 \) be first party and \( l_2 \) second party. After the creation of the pivot \( p \) in a tree \( T(t) \) there will be at least one finalization of a completed synchronization between two leaves \( sl_1 \) and \( sl_2 \) in \( L(p) \) in the tree \( T(p) \).

Proof: The proposition is proved by induction over the depth of tree \( t \).

\[
\text{Figure 8.16: Possible Synchronizations in } T(p)
\]

1. \( D(p) = 1 \). There is no other possibility than that \( l_1 = sl_1 \) and \( l_2 = sl_2 \) and \( \text{Sync}(l_1, l_2) \) succeeded and the proposition clearly holds.

2. \( D(p) = n \) such that \( n > 1 \). First we state the induction hypothesis. For each \( T(p) \) such that \( D(p) < n \) proposition 8.2 holds. To prove the proposition for \( D(p) = n \) four cases occur (see fig. 8.16):

   (a) \( sl_1 = l_1 \) and \( sl_2 = l_2 \) (no local synchronizations): then again \( \text{Sync}(l_1, l_2) \) succeeds and the proposition holds.

   (b) \( sl_1 = l_1 \) and \( sl_2 \in L(C_r(p)) \) (a local synchronization in the first branch during 2L1S): let \( d_1 \) be the depth of the subtree in which
the local synchronization takes place. Then \( d_1 < d \). Upon starting the synchronization under \( p \) phase 2L1S is stopped either on the pivot \( p_1 \) in figure 8.16, or on a free node \( f \). In the first case induction assures that there will be local a synchronization \( \text{Sync}(p_2, l_1) \) under pivot \( p_1 \). In the second case there has been a full synchronization somewhere in \( T(f) \), which is not yet completed and is in the cleaning up phase and has not yet reached \( p \) and again the proposition holds.

(c) \( s_1 \in L(C_1(p)) \) and \( s_2 = l_2 \) (a local synchronization in the second branch during 2L2S): again let \( d_1 \) be the depth of the subtree in which the local synchronization takes place. Then \( d_1 < d \). Unpon starting the synchronization under \( p \) phase 2L1S and 1L2S succeeded and 2L2S is stopped either on a pivot or on a free node. In the pivot case again induction assures that a local synchronization \( \text{Sync}(s_1, l_2) \) will take place. In the case 2L2S stopped on a free node \( f \), then again there has been a full synchronization somewhere in \( T(f) \), which is not yet completed and is in the cleaning up phase and has not yet reached \( p \). So the proposition holds.

(d) \( s_1 \neq l_1 \wedge s_2 \neq l_2 \) but then it must be that there is a pivot \( p' \) such that \( p' \in L(p) \) and \( l_2 \notin L(p') \) and \( l_1 \notin L(p') \). After all, if \( l_2 \in L(p') \) then either \( s_1 \) or \( s_2 \) would have met \( (l_2, p') \) during pebbling and then one of the cases 1. 2. or 3. would apply. As \( [l_1, p] \cap N(p') = \{ \} \) and \( [l_2, p] \cap N(p') = \{ \} \) the whole synchronization \( \text{Sync}(s_1, s_2) \) is independent of the creation of the pivot \( p \). Induction assures that after the creation of \( p' \) a synchronization will occur as \( D(p') < D(p) \). \( \square \)

Proposition 8.2 starts from the fact that a pivot comes into existence. The next proposition states under which conditions a pivot is created.

**Proposition 8.3 (Causality of Pivots)** Let \( S(n) \) be a pebbled spine in \( T(t) \). After the arrival of an active leave \( l \) a pivot \( p_{n+1} \) comes into existence in \( T(t) \) such that \( p \in (S(n) \cap S(l)) \) if there has not been a completed synchronization that involved \( n \) before the moment the pebbling process of \( l \) reaches \( p \). \( \square \)

**Proof:** The binary tree structure ensures that \( \exists p : p \in L(l) \wedge p \in L(n) \). On arrival of the pebbling phase initiated by \( l \) at the node \( p \) two cases can occur:

- \( p \) is free: no pivot is formed and pebbling continues to \( t \). Note that the node \( p \) can be free only if there has been a completed synchronization.
in $T(p)$ that involved $n$ (see figure 8.3).

- $p$ is pebbled and then $p$ is turned into a pivot.

The node $p$ can not already be a pivot or a false pivot. The pebbling phase progresses only over free nodes and thus, upon arrival at $p$ any involvement of $p$ in a synchronization must be terminated completely.

The following proposition ensures that no two processes may become waiting on a partner to arrive. Let $AL_w(n)$ denote the number of active leaves in $T(n)$ that are waiting either in state 2 or 10. First a lemma is stated that is used within the proof of proposition 8.4.

**Lemma 8.3** On the transition moment of a pivot node $p$ with $D(p) = d$ to a node with another action structure value, $AL_w(p) = 0$.

**Proof:** If a pivot $p_{n,l}^{n,l}$ is created then (1.) it will evolve to a free node after a successful synchronization, or (2.) it will be turned into a spine node or a false pivot after a failed synchronization attempt.

1. A free node indicates that no pebbled spines occur any more in $T(p)$ and subsequently $AL_w(p) = 0$.

2. A spine node marks the passage of a repebbling phase belonging to one of the nodes $n$ or $l$. The other node has been involved in the synchronization and is not active any more. Thus $AL_w(p) = 0$ in this case also.

So in each case $AL_w(p) = 0$.

This lemma forms the basis to the final proposition.

**Proposition 8.4 (Exclusion of tandem-deadlock)** It is not possible that two active leaves exist in a tree $T(t)$ such that their P2L3S procedure ended in either state 2 or state 10 of figure 8.5.

**Proof:** Suppose a leave $n$ exists in a tree with top node $t$ that has its spine pebbled and is waiting at state 2 or 10. So $AL_w(t) = 1$. Now a leave $l$ arrives, then proposition 8.3 assures that a pivot $p_{n,l}^{n,l}$ is created when it is due.

1. If no pivot is created on the cross point $p$ of the spines of $n$ and $l$ when the pebbling phase of $l$ reaches $p$ then $n$ has been involved in a synchronization and is not an active leave any more and $p$ must be free. So at the moment the pebbling phase of $l$ crosses $p$ $AL_w(p) = 0$ and $l$ continues pebbling.
2. If a pivot \( p' \) is created somewhere on \([l, p)\) lemma 8.3 assures that 
\( AL_w(p') = 0 \) and \( n \) remains waiting, and so \( AL_w(t) = 1 \).

3. If \( p \) is turned into a pivot then lemma 8.3 ensures that at the end of 
\( p \)'s existence as a pivot \( AL_w(p) = 0 \).

So in each case the number of waiting processes \( AL_w(t) \leq 1 \). \( \square \)

The three propositions together in the reverse order, under assumption of 
correct phase succession, ensure the correctness of the functionality of the 
protocol.

1. Proposition 8.4 excludes situations that two or more active leaves are 
waiting i.e. without creating a pivot.

2. Proposition 8.3 assures that a pivot is created when due.

3. Proposition 8.2 links the creation of a pivot to completed synchrone-
izations.

### 8.6 Extensions to the protocol

This section sketches four simple extensions to the basic protocol.

1. The first extension allows for message passing between synchronizing 
processes. The ability of message passing realizes a partition over 
the set of action labels: one class for output actions and one class 
for input actions. Now processes synchronize on an action label with 
specified direction of message passing: in or out. Two in action labels 
cannot synchronize, equally, two out action labels cannot synchronize 
either.

To model the message passing we let pebbles hold the message. The 
message flows from out-party to in-party by passing it to all pebbles 
that form the path between the two parties:

- first party is an in-party: the message from the second party 
is taken upwards in 1L1S and downwards in 2L1S. After the 
synchronization is acknowledged by a completed 2L3S phase the 
message becomes available to the first party.

- first party is an out-party: the message is taken from the first 
party at the beginning of 1L2S and on reaching the second party 
at the completion of 2L2S it is directly available.
8.6. Extensions to the protocol

The two classes of action labels are represented by coloured pebbles: one colour for pebbles involved with an in-action, and one colour for the out-action pebbles, both sharing the same action-structure. I let a black dot represent in-action pebbles and a dot containing a cross represent an out-action pebble. The set of viable action-structure values now becomes:

- : in-action spine nodes,
- : out-action spine nodes,
- : node claimed by in-action going to out-action leave,
- : node claimed by out-action going to in-action leave,
- : pivot for in-action going to out-action,
- : pivot for out-action going to in-action,
- : false pivots.

Together with the eight action-structure values presented earlier the number of different values has reached 22.

The phases succession of figure 8.5 remain unaltered. Each phase description is extended to incorporate the actions on the added action-structure values: e.g. during PS and 1L1S pebbling is postponed when a node is reached that holds already an equal coloured pebble until it is freed. A pivot is formed when a spine node is reached with opposite direction colour.

2. An extension can visualised to incorporate the choice operator. The effect of a successful synchronization in the context of an operand of the choice operator will cause the termination of all nodes in the subtree of the other operand of the choice. The nodes may act along a first come first serve paradigm, such that in the context of e.g. $a.X + a.Y$ the operand that entered its pebble first annihilates the other operand.

3. (Restricting synchronization) Restricting the spine pebbling phases PS, 1L1S, R1L1S and RS to pass across a specific node $p$ in the tree $t$ inhibits the nodes in $L(p)$ to synchronize with nodes $L(t) \setminus L(p)$. This restriction realizes embedded subsets of $L(t)$ that are only capable of local synchronizations.
4. (Relabelling action-labels) Instead of restricting the pebbling phases to pass across a specific node, the actions can be switched from one action-label to another, thus modelling a relabelling of action-labels.

8.7 Related work

Besides the symmetrical synchronous model that is sketched below there are other synchronization and communication models. A general introduction to the field of distributed computing can be found in the book of BEN-ARI ([26]). In [21] the authors present a survey of programming languages for distributed computing systems, classified by the applied paradigm with respect to inter-process communication. In [12] ANDREWS presents another survey, dedicated to the different paradigms for process interaction in distributed systems.

E.g. the programming language Ada ([217]) employs an asymmetrical model with respect to the inter-process awareness. The synchronization is called a rendezvous and there is always a server process that is unaware of the process it is servicing, and a client process that must know the identity of the server process before it can get any service at all. The data values may pass from one party to the other party, or bi-directionally, from one party to the other and back again. Yet other models start from message passing directly to named processes, either synchronously like in the language Occam [107] or asynchronously like in Concurrent C [215]. Other approaches are e.g. the Remote Procedure Call applied in Cedar [204] and concurrent CLU [88].

There has been other efforts to describe symmetrical synchronizations between processes in a distributed environment, although each of them is dedicated to the implementation of the CSP communication primitives. I have found no contributions that covered the CCS primitives. One of the key publications in this field is [47] in which BUCKLEY and SILBERSCHATZ provide both an elegant implementation algorithm and a taxonomy of four criteria that any efficient algorithm should satisfy. BACK and KURKI-SUONIA build on the work of Buckley and Silberschatz and come to solutions based on different forms of communication: point to point, multi-process handshaking and broadcasting ([18], [19], [17]). Another line of contribution has been provided by RAMESH, who developed a fair implementation for shared actions in the context of global nondeterminism and multiprocess synchronizations ([190]).

Synchronization by phases, as applied in the P2L3S algorithm was first applied by GALLAGER-82 in the context of computing routings in com-
puter networks. Later the work of Lakshmam and Agrawala [132] described the application of phases on problems of consensus in distributed environments. An overview of different approaches in the description of synchronization and control in distributed environments is found in the work of Raynal. His recent book [191] covers three different approaches to algorithms: wave algorithms, algorithms with logic pulses and phase algorithms.

8.8 Conclusions

The concept of symmetrical unawareness (SymUn), in combination with distributed data spaces is not a common paradigm in existing programming languages for distributed computing systems. In the survey article of Balcs, many languages are shortly discussed, however, not a single member of these languages employs the discussed approach. The approach taken in the OCCAM language [107], based on the CSP notation of Hoare [103] comes closely to the SymUn approach but assumes a tightly coupled system of processors serving a fixed number of processes with a shared data space.

The uncommonality is really not so remarkable as it seems. The SymUn approach is derived from the Calculus of Communicating Systems (CCS) of Milner [157] which is not a programming language, but a formal notation based on process algebra, specially dedicated to serve as a specification device for concurrent systems. As far as to my knowledge no *programming languages* are yet based on the CCS approach, which is in itself a remarkable fact. The basic operators of CCS to describe concurrent processes are simple, elegant, and very powerful. Like the *Linda* concept of Carrriero and Gelernter [50] the SymUn concept could be easily added on to existing sequential languages, thus enabling them to operate in the field of distributed computing.

Further research is in progress to study the main characteristics of the P2L3S protocol. The protocol has been incorporated in a test case generator implemented in Ada [209] that accepts binary trees representations and simulates their evolution based on the semantic realization with Ada tasking.

I am working on an axiomatic characterization of the protocol in terms of invariants on the binary tree. It is yet an open question if there are other (known) protocols that support the SymUn approach and improve the worst-case behaviour of the P2L3S protocol, which is sub-linear in the number of processes attached to the tree.

The extensions to the protocol presented in section 8.6 cover the whole range of basic CCS operators but excludes the TCCS extensions. I have not
yet tried to incorporate the time extensions in this protocol for several rea-
ssons. The influence of the distributed execution environment on the effect
of the basic CCS operators is nothing with respect to the influence on the
time dependent behaviour. The management of time in distributed envi-
ronments is a complex matter. Secondly I am working on simpler versions
of the proposed protocol that still retain the same complexity attributes.
Before tackling the problems with respect to the timing extensions I would
like to implement the protocol and interface the runtime support for a well
known programming language for non-distributed programming, such as C.
9.1 Introduction

The last four chapters have basically served to gather information to picture the MOSCA notation within the overall process of software development. The main issue of these chapters was to gain insight in finding answers to the question that was formulated in the introduction of chapter 5 and the topics put in the validation suite. This closing chapter is further organised as follows. Section 9.2 offers an evaluation of the research effort. Section 9.3 presents some pointers into future research. Section 9.4 offers a concluding remark concerning the research itself.

9.2 Validation of the research effort

In this section I will address the validation suite stated in the opening chapter. Further the question stated in the introduction of chapter 5 is taken up to investigate the conjectured answer from the closing section of chapter 5 and finally I shall address the hypothesis of the thesis in order to decide either to accept or to reject it.

9.2.1 Validation of MOSCA

9.2.1.1 The semantics of MOSCA

| Is the combination notation based on a complete and sound theoretical base? |

The semantics of MOSCA are still unfinished. Although the dynamic semantics are complete, the static semantics are totally ignored in this thesis. The reason for this negligence is simply a matter of time and natural inclination.
My personal affection to language definition has always been stronger for the questions concerning what the meaning of the language constructions should be rather than to the problem of finding out which syntactical correct phrases can be given a meaning. Whereas the latter problem may not be too difficult to solve for standard programming languages it is certainly very complex for languages such as VDM-SL. Our group has investigated the problem to a certain extend ([176]) out of practical motives. The VDM-SL front-end compiler, which has been under development for several years, facilitates a certain level of type checking. Although the matter is partially solved Hans Bruun has shown in a substantial research effort that finding a complete solution for this problem is very difficult ([46], [60]).

I have limited my research to the dynamic semantics purely on pragmatical grounds. The outcome of the ongoing research of Bruun was not yet fixed during the time I started to work on MOSCA, whereas the structure of the dynamic semantics was more or less stable. During the evolving research I became involved with too many other important features of the language to pick up the work of Bruun as well. Another important factor has been the cooperation with Peter Larsen, who has visited Delft during a period of several months. Peter was the main implementor of the dynamic semantics and provided me with the necessary background to continue my work. It has never been my intention to finish the whole language definition of MOSCA within the time span of the research topic of this thesis. This work is clearly beyond the grasp of a single person. The dynamic semantics forms the kernel of all analysis techniques and as such it had to be defined up to a satisfactory level of detail, in which I feel to have succeeded.

A point of concern in this matter remains the overall operational quality of the current approach. The operational semantic setting found in the SOS approach is severely limited by the associated properties of the VDM-SL semantics. The denotational semantic definition for VDM-SL is certainly far from being operational. Even for trivial specification fragments such as a simple type definition like

\[
\text{naturals} = \mathbb{N}
\]

all possible environments \( env \in VENV \) are checked, i.e. environments containing all possible different type denotations, function denotations, operation denotations, etc. Through the \( \text{IsAModelOf} \) predicate all environments are rejected but one. Only the model containing a proper denotation for the natural numbers is passed and nothing else. Even if it would be possible to enumerate all environments explicitly, it would take infinite time to check them all. Clearly one has to restrict the VDM-SL notation to something more modest and give a characterization of this subset in a more opera-
9.2. Validation of the research effort

Tional style to be able to define something really operational. E.g. Larsen has defined a subset of VDM-SL operationally ([137]) that retains looseness within expressions and patterns.

After having studied the dynamic semantics of the VDM-SL and the power of expression of structural operational semantics in general, I consider it an isolated research topic to investigate whether it is possible to construct a structured operational semantics for the whole VDM-SL language and as such for the whole MOSCA language.

9.2.1.2 The language that resulted from the combination

| Has the combination notation retained the important properties of the constituent parts? |

Here the answer is both affirmative and negative! To ground the answer to this question different aspects of both constituents must be addressed.

The VDM-SL part has been incorporated completely and has retained all of its specific qualities with respect to specification, refinement, implementation and verification.

Regretfully this cannot be stated from several qualities of the incorporated TCCS parts. The TCCS notation has been given different characterizations, from which the SOS and the equational setting are well known. The notions of similarity, briefly introduced in section 2.3.3, have not been touched upon in this research. The effects of VDM-SL refinement is not easily mixed with the notions of similarity of TCCS. A brief example here will stress my point. Assume two VDM-SL functions \( f: In \rightarrow Out \) and \( f': R_{\text{In}} \rightarrow R_{\text{out}} \) given such that \( f' \) has been shown to be an adequate refinement of \( f \). Now assume the two agents \( A \) and \( A' \)

\[
A \langle x \rangle \triangleq \text{output}(f(x)) \odot P
\]

\[
A' \langle rx \rangle \triangleq \text{output}(f'(rx)) \odot S \odot P
\]

such that \( P \sim S \odot P \) under the same definition of similarity. Now what can be said from the two agents \( A \) and \( A' \)? Similarity induces the fact that each involved agent can simulate the behaviour of the other. But their behaviour is certainly not equal in the sense that \( f \) computes something totally different than \( f' \). I have the strong intuition that a whole new concept is evolving here, something not quite identical to similarity. At the moment it is certainly not understood by me in acceptable detail to have it covered within the thesis.

There are several other issues that are not addressed as well, e.g. the expansion law of CCS. The impact of the time extensions in combination
with VDM-SL looseness with respect to this topic is currently beyond my understanding. Again I must remark that further research on this topic is needed to clarify the issues.

On the positive side there are fortunately enough qualities of CCS retained in the combination. The most important aspect retained is the SOS setting. Thus the basic proof system for CCS is still available and forms the basis for analysis approaches such as state space analysis and trace analysis. Due to the orthogonality of the combination particular aspects of MOSCA specifications can be reduced to be either induced by VDM-SL or CCS. As such the similarity relations and expansion laws can still be applied to derived forms of MOSCA specifications from which the relevant parts are extracted.

9.2.1.3 The applicability of MOSCA

Is the combination notation a possible candidate for fruitful application within the activities for TSCS development?

To answer this question the five associated topics are addressed as well as the question raised in the introduction of the fifth chapter.

Given the particular view on software development contained in the software development framework, where does MOSCA fit in?

I feel that the final outcome of the quest of placing MOSCA within the software development framework is not yet resolved. In the two case studies MOSCA is used both as software requirement specification language and system modelling language. In the first case study the VDM-SL data modelling capabilities enabled a concise high level specification which was transformed into a more concrete specification due to the CCS capabilities of composition and communication. In the second case study the state based nature of the controller was elegantly combined with the concurrency aspects that were inherently present in the subject of choice. However, the presentation of the case studies have not been subjected to subsequent refinement and implementation of the specifications. And as such no experience of using MOSCA in the latter phases of software development has been gained.

Further I must admit that the presentation of only two case studies in itself forms a too narrow basis for deciding on the applicability of MOSCA. Much more experience is needed to give a definite answer to the stated question. The observations conjectured in the closing section of chapter 5 still express matters of concern. The impotence of MOSCA with respect
9.2. Validation of the research effort

to the formulation of timing constraints on an acceptable level of abstraction strongly suggest the fact that other techniques should be used besides MOSCA to cover the non-behavioural requirements of TSCS's.

Summing up this analysis I think it is at least fair to decide that MOSCA offers a potential strong tool to describe the behavioural constraints of TSCS's. This conclusion is strengthened by skimming through the list of qualities of specification languages presented in sections 5.3.3.1 and 5.3.3.2.

<table>
<thead>
<tr>
<th>Is the language expressive enough to specify software components of TSCS's?</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the light of the last analysis the answer to this question must be carefully affirmative, with the addition that the specification is restricted to the behavioural aspects of the software components. There may be important aspects of certain subclasses of the TSCS's that ask for specialised language constructions. An example in this context may be found in the safety critical systems. These kind of systems maintain often a certain degree of fault tolerance and have build in safety precautions that anticipate exceptional behaviour. MOSCA has no specialised language features that support directly exceptional behaviour or fault tolerance. Although the VDM-SL notation is equipped with a simple exception mechanism, there is no support for this feature beyond the scope of the operation in which the exception occurred. I have no strong intuitions on the semantic realization of such mechanism in the context of MOSCA. The CCS operators have been extended with some kind of exception operator (e.g. in the language LOTOS), but the offered service through this operator is in my opinion rather limited. On this specific topic more research is clearly needed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Does the language support formal reasoning necessary for the validation of software components of TSCS's?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Here the answer is certainly positive. The proof system of VDM-SL associated with the dynamic semantics, in combination with the SOS induced proof system offers enough formal power for validation.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Does the language preserve the implementation freedom needed to meet the non-functional requirements of the software components of TSCS's?</th>
</tr>
</thead>
</table>
This is a basic issue that remains unsettled. The analysis concerning the
effect of the CCS primitives on the implementation offered in section 5.4.2
revealed the important impact of abundance of processes on the implementa-
tion structure within the context of a simple processor. In the context
of distributed systems the situation is different. The P2L3S protocol may
form a suitable means to extend existing implementation languages with the
SymUn concept. These extended languages may be capable of implement-
ing MOSCA specifications within the constraints set by the non-functional
requirements. To get a better impression on the quality of the SymUn
approach the experiment should be tried at least once.

<table>
<thead>
<tr>
<th>Is there a method for construction, validation and implementation of specifications in the language?</th>
</tr>
</thead>
</table>

There are in fact two methods here: the VDM approach and the col-
lection of paradigms that can be applied on the CCS specifications. As
remarked above certain aspects of CCS model checking have been invalu-
dated, but most techniques still apply. I have not worked on a possible
merger of the two methods, although I have the strong feeling that the
merger of the two languages should be accompanied by the construction of
a method for application. But I feel that this work is beyond the scope of
the capacity of one person working on this topic. Much more experience
on the application of MOSCA is needed to devise a concise method. The
summery of the approach taken in the home heater example could be taken
as the beginning of an emerging method for the specification of process
control systems.

<table>
<thead>
<tr>
<th>Is the use of the language cost-effective for the construction of software components of TSCS’s?</th>
</tr>
</thead>
</table>

The answer on this topic depends clearly on the answers of putting the
same question up in the context of VDM-SL and CCS. I would say that
the latter two answers are certainly positive, thus providing an affirmative
answer on this last topic.

### 9.2.2 The research hypothesis revisited

The last issue to address remains the research hypothesis: "Is the merger
of VDM-SL and CCS valid after all?" A brief summery of the outcome of
the research effort is presented on which ground we may decide whether to
accept or reject the hypothesis.

1. A language is defined based on VDM-SL and CCS extended with
capabilities to model time dependent behaviour by:
9.3. Future research

(a) the definition of a mathematical concrete syntax (the syntax used within this thesis) for the language (given in appendix A),

(b) the definition of an ascii version of the mathematical concrete syntax for the language (defined in [170]),

(c) the definition of an abstract syntax that forms the basis for the formal semantics,

(d) the definition of a formal semantics based on the dynamic semantics of VDM-SL and the SOS setting for CCS.

2. The orthogonality of the approach has retained the availability of both the standard VDM-SL refinement and proof techniques as well as most of the analysis capabilities of CCS. In addition to these techniques two techniques dedicated to the whole range of the MOSCA language are proposed and investigated to a certain extend: state space analysis and trace analysis.

3. The SymUn has been proposed to provide a possible basis for the implementation of MOSCA specifications in a distributed environment.

On these grounds it must be fair to state that the research hypothesis must be accepted.

9.3 Future research

This research has opened up several unfinished paths of work. The topics are arranged in four groups, concerning the language, specification, analysis and implementation.

9.3.1 Matters concerning the language

During the work on MOSCA I have conceived the idea of incorporating an applicative flavoured style of process specification that deviated strongly from the original CCS style. The specific form of a process resembled strongly the stream notion that is found in the work of many researchers working in the field of concurrency in a functional style. The ideas were strongly inspired by the early work of KAHN ([119], [120]) and HENDERSON ([92], [93]). More recent work on these topics can be found in the work of BURTON ([49]), GIACALONE ([79]), KELLY ([126]) and TURNER ([216]). I like to present a simple example of the extensions in mind to give an idea of the possible forms of a stream agent.
Example 9.1 E.g. Suppose we have defined a mixer agent that mixes the values of its two input streams into one output stream according to specification 9.1. The stream type \( N^\infty \) specifies a stream of natural numbers, generated by a stream agent. In the example a special case is presented, the \((\quad)\) form, that denotes the empty type, signifying a stream with no input ports at all. The construction following the \(\mapsto\) symbol specifies the type of values that can be passed along the single output port of the agent.

We could mix the output of the mix agent itself, like in

\[
\text{let } x = \text{mix}(\text{NaturalsFrom } 1, x) \text{ in } \ldots
\]

This recursive let construction defines the stream of naturals starting from 1, given by

\[
1, 1, 2, 1, 3, 2, 4, 1, 5, 3, 6, 2, 7, 4, 8, 1, 9, 5, 10, 3, 11, 6, 12, \ldots
\]

The sequence of naturals resulting from the stream \(x\) can be given in a closed formula. Expressed as a series \(z_n\), it is defined by:

\[
z_n = \begin{cases} 
  (n + 1)/2 & \text{when } n \text{ is odd} \\
  z_n/2 & \text{when } n \text{ is even}
\end{cases}
\]

Substituting \(x\) within the definition gives

\[
\text{Mix(\text{NaturalsFrom } 1,} \\
\text{Mix(\text{NaturalsFrom } 1,} \\
\text{Mix(\text{NaturalsFrom } 1,} \\
\text{\ldots)))}
\]

which corresponds with the flow graph given in figure 9.1. The let expression defines a dynamic structure. Each successive put action of the \(\text{Mix}\) agent enwinds the flow graph one step further. \[\square\]
9.3. Future research

The meaning of the let construction defining the stream \( x \) can be given according to various models. One particular model views the meaning of the let construction as the solution of the definition

\[
x = F(x)
\]

where the function \( F \) is derived from the \textit{Mix} function. A solution to this definition is a fixed point of the function \( F \). These fixed points exist only if the functions they are associated with are continuous with respect to their definition domain and this domain is a chain-complete partial ordering. This interpretation for the let construction needs a full denotational model for streams, and subsequently a full denotational model for the whole agent sublanguage of MOSCA.

The definition of such denotational model is a possible candidate for future research. It can be based on the denotational models of Hennessy (see e.g. [95] and [94]).

9.3.2 Matters concerning specification

The two case studies form a too narrow base to serve as tests for the development of an associated MOSCA development method. Much more experience must be recorded before such method can be constructed.
9.3.3 Matters concerning analysis

The state space analysis offers a wide range of derived state spaces to serve during analysis. Here future research is needed to find elegant algorithms to generate and subsequently search the derived state spaces. The notion of partial derived state spaces must be clarified and defined properly.

The analytical power of trace analysis is not yet clear. The definition of traces over infinite behaviours must be thought over to find out if a more attractive representation can be devised than the union approach stated in section 6.3.

The MOSCA rapid prototyper is still in an experimental stadium. Currently a student is working on the design of a MOSCA kernel that offers a C interface, and after a reworking of the existing code generator to get C code instead of Ada code, is is planned to serve as runtime supervisor for the rapid prototyper.

9.3.4 Matters concerning implementation

This topic still needs much attention. I am very interested in the continuation of the work on the P2L3S protocol and SymUn approach. The protocol should be extended to include the time aspects of MOSCA as well. The first real experiment could be the definition of a C language extension.

9.4 To conclude

This research has been conducted starting from the goal to find a means to specify TSCS’s by means of cooperation of VDM-SL and TCCS in some specific form. The accent clearly has been put on the form aspect and resulted in MOSCA.

A totally different setting to conduct the research to find a specification language for TSCS’s could start with the definition of a set of criteria on which properties for candidate specification language features could be selected. The selected properties could then form the basis for subsequent language component construction.

I feel that in some aspects my approach is advantageous over the second but in other aspects the second approach could come up with better results. The advantages are clearly the vast amount of imported work in the form of the syntax, semantics and pragmatic issues of both VDM-SL and TCCS. The disadvantages come forth out of the same reason. The actual setting of the syntactical constructions and their semantics limit the amount of freedom to realize certain properties. This fact has been clearly illustrated
9.4. To conclude

by the mismatch between the formulation of the timing constraints and the realization of these constraints within MOSCA.
Presentation syntax

A.1 Introduction

The following syntax specification is aimed at providing a firm basis for the presentation (in mathematical font) of the MOSCA specifications within this thesis. Syntax specifications tend to act as both arbiter and straitjacket. They e.g. decide on what is syntactically right and what is syntactically wrong, which is a positive aspect of syntax. But they also enforce a precise usage of the specified notation, without providing any freedom in presentation at all. When using mathematical constructions within a notation, it can be very elegant to use no precise syntax at all, and to be free to introduce notational devices whenever the need arrives. This approach is followed in e.g. [157], where Milner describes the capabilities of his agents with respect to value manipulation in a completely free manner, introducing (very powerful) notation whenever it seems fit. Chandy in [52] proceeds in similar style. Again no syntax is given for the unity notation, although each construction is presented in a clear way. Even the VDM notation is sometimes used in a free manner. In [146], this "abuse of notation", is even considered a blessing. To quote Mac an Airchinnigh ([146], p. 143) "Abuse of notation is a fundamental part of mathematical culture. It is a deliberate abandoning of the strictly precise and formal denotation of mathematical entities." He claims that the liberal mathematical approach opens the door to a potential of fruitfull ambiguity. In his thesis [145] he treats a new variant of the VDM, denoted "The Irish School of the VDM". Without presenting any formal syntax or even a formal semantics, he succeeds in a clear and meaningful presentation based on constructive mathematics.

In my opinion, the latter view is elegant and convincing, but unsufficient whenever the notation is going to be supported by automated tools. It is for the usage of the toolbuilder that formal syntax is meant. Although my
notation is clearly not ready at all for usage as keyboard language, it is a notation that can be easily converted to a notation based on the ascii character set, without abandoning the syntactical structure presented below. As such the presentation syntax may serve as a basis for tool development. The primary distinction between the two notations is in the available character sets: for the mathematical syntax we will take advantage of a variety of non-ASCII characters and representations.

The current choice of symbols within the presentation syntax is strongly influenced by both the VDM and CCS notations and the document preparation system used to produce this text, \textbf{LaTeX} [134].

I have strongly used the aspects of visualization. Constructions in a specification are often separated from each other by placing the construction on a single line, without adding explicit separators as ‘;’, ‘;’ etc. The separators are omitted in the most places. This causes the presentation syntax certainly not to be (La)LR.

\textbf{mosca} is not a small notation. This is merely a result of the size of the embedded notation of the VDM part, which is rather rich in syntactical presentation. The two sections in this chapter describe the agent language which has been put on top of the VDM notation. The VDM notation is not described. The notation for VDM-SL is currently fixed on a level of abstract syntax. The mathematical concrete syntax of VDM-SL is currently being under investigation of an ISO standardization review board\footnote{Under guidance of ISO SC22/WG19}. The abstract syntax serves as basis for the semantics of \textbf{mosca-SL}.

\textbf{Notation}

The proposed concrete representation for the \textbf{mosca} specification language is defined by a context-free grammar. The grammar is described by means of an extended BNF notation which employs the following special symbols:

\begin{itemize}
  \item \textbf{,} the concatenate symbol
  \item \textbf{=} the define symbol
  \item \textbf{|} the definition separator symbol
  \item \textbf{[ ]} enclose optional syntactic items
  \item \textbf{\{} \textbf{\}} enclose syntactic items which may occur zero or more times
  \item single quotes are used to enclose terminal symbols
\end{itemize}
A.2. Collection Specification

**meta identifier** non-terminal symbols are written in lower-case letters
(possibly including spaces)

; terminator symbol to denote the end of a rule

( ) used for grouping, e.g. a,(b | c) is equivalent to a, b | a, c

- denotes subtraction from a set of terminal symbols

The precedence of the metalanguage symbols is

<table>
<thead>
<tr>
<th>precedence level</th>
<th>metalanguage symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>,</td>
<td>concatenate</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>or</td>
</tr>
</tbody>
</table>

The syntactical notions are annotated with some general remarks.

A.2 Collection Specification

There is no strong structuring mechanism in MOSCA-SL. A module structure is not present. The reason for this omission is given in chapter C in the thesis. The only structuring notion is the encapsulation of a set of definitions through a collection.

**PS-1** A collection consists of an ordered set of items. An item is the definition of a type, value, function, agent or unit. A such, collections can be completely (i) value-oriented, e.g. ADT's, (ii) agent-oriented, e.g. built from server agent definitions or (iii) mixed. Mark that collections cannot define state.

**PS-2** Items have unique names.

**PS-3** All items are to be exported.

**PS-4** Collections are flat, i.e. they cannot import anything.

**PS-5** The outermost scope construction in MOSCA is a specification. A specification is built from (i) a series of unit-import clauses, (ii) a list of local definitions and (iii) a specification-expression, i.e. an agent, in which all agent and unit definitions defined in the specification can be composed.

**PS-6** Items can be imported into specifications by either explicit item import constructions, or collection importation resulting in all items imported together.
1. collection = ‘collection’, identifier, ‘is’, definitions ;
2. definitions = definition block, { definition block } ;
3. definition block = type definitions
   | value definitions
   | function definitions
   | agent definitions ;
4. specification = ‘specification’, identifier, ‘is’,
   [ import specification ],
   [ definitions ],
   behaviour expression ;
5. import specification = item import specification series
   | collection import ;
6. item import specification series = item import specification,
   { item import specification } ;
7. item import specification = ‘from’, identifier, ‘import’, identifier ;
8. collection import = ‘from’, identifier, ‘import all’ ;

A.3 Agent Specification

A.3.1 Agent Definition

PS-1 Agents occur in two styles, one style applying an explicit composing rule and another style using a more functional approach to agent composition. The latter agents are named streams.

PS-2 Ordinary agent definitions can be simple or compound.

PS-3 A simple agent definition consists of an agent heading, a port specification, one or more agent behaviour definitions and possibly a series of local definitions. The agent behaviour definitions all have (i) the same name, i.e. the name of the agent and (ii) share the same heading and port specification. The agent heading and port specification is stable during the lifetime of the agent.

PS-4 A compound agent, called a unit, also defines one agent, but now the agent’s heading and port specification may vary during it’s lifetime. The unit heading and port specification is refined by the series of agent behaviour definitions. Each agent behaviour definition may extend the port specification of the unit, without altering the original ports. Furthermore, each agent behaviour definition may replace the original value part specification. The state part specification in the unit heading is stable, visible during the whole lifetime of the agent.
A.3. Agent Specification

9. agent definitions = unit or agent, ‘;’, { unit or agent } ;
10. unit or agent = unit definition | agent definition ;
11. unit definition = ‘unit’, agent heading, port specification,
   [ share list ],
   { agent behaviour },
   [ local definitions ] ;
12. share list = share list item , ‘;’, { share list item } ;
13. share list item = identifier, [ vpart specification ],
   [ port specification ] ;
14. agent definition = agent heading, port specification,
   { agent behaviour }, [ local definitions ] ;
15. agent heading = identifier, [ vpart specification ],
   [ spart specification ] ;
16. vpart specification = ‘{’, type, ‘}’ ;
17. spart specification = ‘state’, field list,
   [ spart invariant ],
   [ spart initialization ] ;
18. port specification = ‘ports’, port typelist ;
19. port typelist = port type, { port type } ;
20. port type = syn porttype
   | in porttype
   | out porttype ;
21. syn port type = ‘syn’, identifier ;
22. in porttype = ‘in’, identifier, ‘;’, type ;
23. out porttype = ‘out’, identifier, ‘-’, ‘;’, type ;
24. spart invariant = ‘inv’, identifier, invariant initial function ;
25. spart initialization = ‘init’, identifier, invariant initial function ;
26. agent behaviour = identifier,
   [ vpart pattern ],
   ‘≜’, behaviour expression ;
27. vpart pattern = ‘{’, pattern, ‘}’ ;
28. local definitions = ‘where’, agent local definition,
   { ‘;’, agent local definition } ;
29. agent local definition = type definitions
   | value definitions
   | function definitions
   | operation definitions²;

²Issue: Should agents be added to this list?
A.3.2 Agent Behaviour

30. behaviour expression = agent service
   | agent if
   | agent let
   | agent let be
   | prefix
   | choice
   | composition
   | restriction
   | relabelling
   | null agent ;

31. agent service = name, [ vpart expression ] ;

32. vpart expression = ‘{’, expression, ‘}’ ;

33. agent if = ‘if’, agent if expression,
     ‘then’, behaviour expression,
     ‘else’, behaviour expression ;

34. agent let = agent value let | agent agent let ;

35. agent value let = ‘let’, let local definition,
     ‘in’, behaviour expression ;

36. let local definition = type definitions
   | value definitions
   | function definitions ;

37. agent agent let = ‘let’, identifier, ‘=‘, behaviour expression,
     ‘in’, behaviour expression ;

38. agent let be = ‘let’, pattern bind,
     [ ‘be’, ‘s.t.’, expression ],
     ‘in’, behaviour expression ;

39. agent if expression = expression ;

40. prefix = action, ‘⊙’, behaviour expression ;

41. action = syn action
   | in action
   | out action
   | spart manipulation ;

42. syn action = identifier, [ time variable binding ] ;

43. in action = identifier,
     ‘{’, pattern, [ duration specification ], ‘}’,
     [ time variable binding ] ;
A.4. Value Specification

44. out action = identifier^3, '→',
    ('(', expression, [ duration specification ],')',
    [ time variable binding ] );

45. spart manipulation = 'σ(', statement, [ duration specification ] );

46. duration specification = '→', 'σ' expression ;

47. time variable binding = '→', 'σ' identifier ;

48. choice = behaviour expression, '⊕', behaviour expression ;

49. composition = behaviour expression, '|', behaviour expression ;

50. restriction = behaviour expression, '\', '{', port list, '}' ;

51. port list = identifier, '{', identifier } ;

52. relabelling = behaviour expression, '[', port map, ']' ;

53. port map = port maplet, '{', PortMaplet } ;

54. port maplet = identifier, '→', identifier ;

55. null agent = 'null' ;

A.4 Value Specification

For the value specification in MOSCA-SL the mathematical syntax of the
VDM-SL is closely followed.

PS-1 The module description part has been deleted and replaced with the collection\textsuperscript{PS} construction.

PS-2 VDM-SL state definitions are deleted. Only the spart specification from the agent definition defines state. Subsequently the VDM-SL state invariant and initial functions are also deleted and replaced with resp. spart invariant and spart initialisation.

PS-3 Operations can only appear within the local definition part of agent definitions that have an associated spart specification. Operations without state-access and with a result value are thus abandoned from agents without state.

\textsuperscript{3}the identifier should be marked by a bar over the name
Core Abstract Syntax

B.1 Introduction

This chapter presents a core abstract syntax (CAS) for MOSCA-SL. It is presented in plain VDM-SL domain definitions. Each domain defines a class of abstract syntax constructions. Each construction is annotated with some general remarks. The CAS is the starting point for the definition of the semantics in the next chapter.

The transformation from the presentation syntax to the core abstract syntax is for a large part straightforward. For the VDM-SL it is formally defined as part of the VDM standard. For MOSCA-SL the mapping is intuitive and explained where needed.

B.2 Collection Specification

\[
\begin{align*}
Spec &::= \text{defs} : \text{Collection} \\
expr &::= [BExpr]
\end{align*}
\]

AS-1 A specification consists of a collection of definitions and an optional behaviour expression. It is assumed that all import clauses that appear in the concrete syntax equivalent of the abstract specification, together with the local definitions are assembled into one collection, that represents all defined entities. As such the dynamic semantics are defined w.r.t. this abstract syntax node as starting point.

\[
\begin{align*}
\text{Collection} &::= \text{valuem} : \text{VDMGlobals} \\
\text{unitm} &::= \text{Id} \xrightarrow{m} \text{UnitDef} \\
\text{agentm} &::= \text{Id} \xrightarrow{m} \text{AgDef}
\end{align*}
\]
Appendix B. Core Abstract Syntax

AS-2 A collection consists of a number of groups of different definitions. The
groups consist of VDM types, explicit function definitions, implicit
function definitions, value definitions, and unit definitions, agent and
stream definitions, so no state definitions.

\[
\begin{align*}
VDMGlobals & ::
\text{typem} : Id \xrightarrow{m} \text{TypeDef} \\
\text{expofnm} : Id \xrightarrow{m} \text{ExpPolyFnDef} \\
\text{exmofnm} : Id \xrightarrow{m} \text{ExpFnDef} \\
\text{impofnm} : Id \xrightarrow{m} \text{ImplPolyFnDef} \\
\text{immpofnm} : Id \xrightarrow{m} \text{ImplFnDef} \\
\text{valuem} : Id \xrightarrow{m} \text{ValDefs}
\end{align*}
\]

B.3 Agent Specification

B.3.1 Agent Definition

\[
\begin{align*}
\text{UnitDef} & ::
\text{vpart} : [\text{Type}] \\
\text{ports} & : \text{PortDef} \\
\text{share} & : Id \xrightarrow{m} \text{UnitItem} \\
\text{localdefs} & : \text{Definitions}
\end{align*}
\]

AS-1 The unit heading (and agent heading) consists only of a value part
type. The state part is moved to the \text{localdefs} construction, which
matches the \text{Definitions} construction from the core abstract syntax
of VDM-SL where upon the dynamic semantics of VDM is defined.
A state definition consists of a type and an initialisation function.
The state invariant is recorded at the type definition of the state.

\[
\begin{align*}
\text{UnitItem} & ::
\text{vpart} : [\text{VType}] \\
\text{pdef} & : [\text{PortDef}] \\
\text{behaviour} & : \text{AgBehaviour}
\end{align*}
\]

\[
\begin{align*}
\text{Definitions} & ::
\text{typem} : Id \xrightarrow{m} \text{TypeDef} \\
\text{expofnm} & : Id \xrightarrow{m} \text{ExpPolyFnDef} \\
\text{exmofnm} & : Id \xrightarrow{m} \text{ExpFnDef} \\
\text{impofnm} & : Id \xrightarrow{m} \text{ImplPolyFnDef} \\
\text{immpofnm} & : Id \xrightarrow{m} \text{ImplFnDef} \\
\text{valuem} & : Id \xrightarrow{m} \text{ValDefs} \\
\text{explop} & : Id \xrightarrow{m} \text{ExpOpDef} \\
\text{implop} & : Id \xrightarrow{m} \text{ImplOpDef} \\
\text{state} & : [\text{StateDef}]
\end{align*}
\]

\[
\begin{align*}
\text{AgDef} & ::
\text{vpart} : [\text{Type}] \\
\text{ports} & : \text{PortDef} \\
\text{behaviour} & : \text{AgBehaviour}^* \\
\text{localdefs} & : \text{VDMDefs}
\end{align*}
\]
B.3. Agent Specification

\[ \text{PortDef :: ports : } Id \overset{m}{\rightarrow} (\text{Cap} \times [\text{VType}]) \]
\[ \text{Cap} = \{\text{SYN}, \text{IN}, \text{OUT}\} \]
\[ \text{Ag Behaviour :: valpattern : } [\text{Pattern}] \]
\[ \text{body : BE Expr} \]

B.3.2 Agent Behaviour

\[ \text{BE Expr} = \text{UnitService} \mid \text{Agent Service} \mid \]
\[ \text{AgentIf} \mid \text{Agent Let} \mid \text{Agent Let Be} \mid \]
\[ \text{Prefix} \mid \text{Choice} \mid \text{Composition} \mid \text{Restriction} \mid \text{Relabelling} \mid \]
\[ \text{Null Agent} \mid \text{Id} \]

\[ \text{Unit Service :: name : } Id \]
\[ \text{itemid : } Id \]
\[ \text{val : } [\text{V Expr}] \]

\[ \text{Agent Service :: name : } Id \]
\[ \text{val : } [\text{V Expr}] \]

\[ \text{Agent If :: test : V Expr} \]
\[ \text{cons : BE Expr} \]
\[ \text{alt n : BE Expr} \]

\[ \text{Void Test :: s argument : } Id \]

\[ \text{Agent Let} = \text{Agent Value Let} \mid \text{Agent Agent Let} \]

\[ \text{Agent Value Let :: locals : Local Def} \]
\[ \text{in : BE Expr} \]

\[ \text{Local Def :: vals : Pattern} \overset{m}{\rightarrow} \text{Val Def} \]
\[ \text{expl fn s : } Id \overset{m}{\rightarrow} \text{Expl Fn Defs} \]
\[ \text{impl fn s : } Id \overset{m}{\rightarrow} \text{Impl Fn Defs} \]

\[ \text{Agent Agent Let :: agents : } Id \overset{m}{\rightarrow} \text{Ag Def} \]
\[ \text{in : BE Expr} \]

\[ \text{Agent Let Be :: bind : Bind} \]
\[ \text{st : } [\text{V Expr}] \]
\[ \text{in : BE Expr} \]

\[ \text{Prefix :: act : Action} \]
\[ \text{res : BE Expr} \]

\[ \text{Action} = \text{Syn Act} \mid \text{In Act} \mid \text{Out Act} \]
\[ \text{State Manipulation} \]

\[ \text{Syn Act :: label : } Id \]
\[ \text{d expr : V Expr} \]
\[ \text{t var : } Id \]
Appendix B. Core Abstract Syntax

```
InAct ::  label : Id
         pattern : Pattern
dexpr  : Vexpr
tvar   : Id

OutAct ::  label : Id
           result : VExpr
dexpr  : Vexpr
tvar   : Id

SM ::  effect : Statement
dexpr  : Vexpr
tvar   : Id

SM ::  action : Statement

Choice ::  lop : BExpr
          rop : BExpr

Composition ::  lop : BExpr
                rop : BExpr

Restriction :: lop : BExpr
               rop : Id-set

Relabelling :: lop : BExpr
              rop : Id \rightarrow Id

NullAgent ::  val : null
```

B.3.3 Auxiliary Domains

This domain is an auxiliary domain that does not reflect a syntactical MOSCA construction but appears in the semantic rules in the definition for the idle prefix.

```
Cdt ::  clockvalue : TIME
       agent : BExpr
```

AS–2 A Cdt resembles a count-down counting device that holds a clock-value and a continuation behaviour. Whenever the clock-value, which holds values from the semantic domain TIME reaches 0 the construction continues its behaviour like agent.
Semantic Rules

The semantic rules fall in three main classes: the environment manipulating rules, the action rules and the idle rules. All environment manipulating rules consist of a semantic function that computes a set of sts configurations and a rule template that defines an internal transition. The actual rules are those defined by the template by quantification over the sts set.

The action rules come in two groups, loose and standard action rules. The actions $x, y$ range over external actions, and $\alpha, \beta$ range over external actions and idle actions. The loose action rules are the agent-if rule and the prefix-in and prefix-out rules. The standard action rules are prefix-syn, choice, composition, restriction and relabelling.

The idling rules contain the idling prefix rules, idling choice, idling composition, the rules for the idle action prefix, the null agent and divergent agent.
C.1 Loose environment manipulation rules

Rule C.1 Specification

\[
\begin{array}{|c|}
\hline
\text{rules Spec} \\
\hline
1 \quad \mathcal{S}(\text{mk-Spec(defs, expr), }\{\}) \triangleq \\
2 \quad \text{let } \text{coll-env-s} = \forall \mathcal{S}[\text{mk-Definitions}(
3 \quad \quad \text{defs.valuem.typem}, \\
4 \quad \quad \text{defs.valuem.expofnm}, \\
5 \quad \quad \text{defs.valuem.ezmoefnm}, \\
6 \quad \quad \text{defs.valuem.impofnm}, \\
7 \quad \quad \text{defs.valuem.immofnm}, \\
8 \quad \quad \text{defs.valuem.valuem}, \\
9 \quad \quad \{\}, \{\}, \{\}) \text{ in} \\
10 \quad \{(expr, \text{mk-Environment}( \\
11 \quad \quad \text{coll-env, coll-env,} \\
12 \quad \quad \text{defs.agentm }\cup\text{ defs.unitm}, \\
13 \quad \quad \text{defs.agentm }\cup\text{ defs.unitm, nil} ) | \\
14 \quad \quad \text{coll-env }\in\text{ coll-env-s} \})
\hline
\end{array}
\]

\[\forall (expr, \rho) \in \mathcal{S}(\text{mk-Spec(defs, expr, }\{\})).\]

C 15 \quad (\text{mk-Spec(defs, expr), }\{\}) \rightarrow (expr, \rho)\]

Annotations to Rule Spec:

There is no hypothesis. The rule transforms the specification construction into a set of state transition states. Each state consists of the behavior expression of the specification and an environment \(\rho\) that reflects one particular semantic evaluation of the definitions.

2–8 The possible models for the collection of definitions that are used within the specification are constructed. On the transformation from the presentation syntax to the abstract syntax all imported definitions are collected within the single collection item \(\text{defs}\).

9 There are no operations and state definitions on this level.

10–14 For each model in \(\text{coll-env-s}\) a MOSCA environment is created, consisting of a model for the VDM definitions \(\text{coll-env}\), the defined agents, the defined units, the defined streams and no state-value identifier. The initial value environment \(\text{coll-env}\) is taken as first local environment.

15 Each environment \(\rho\) forms the basis of an internal transition.
C.1. Loose environment manipulation rules

Rule C.2  AgentService with value expression

\[
\text{rules AS-ve}
\]

1. \( SE(\text{mk-Agentservice}(N, \text{expr}), \rho) \triangleq \)
2. \( \text{let } env = \rho.Cenv, agdef = \rho.Gaenv(N) \text{ in } \)
3. \( \text{let } ibenv-s = LP(\{(i, be(env)) \mid (i, be) \in \)} \)
4. \( \{eeval(env) \mid eeval \in VS[expr]\}) \mid (i, pateval) \in \)
5. \( LP(\{(ind, VS[bh.valpattern]) \mid \)} \)
6. \( \text{ind} \in \text{inds agdef.behaviour} \land \)
7. \( \text{bh = agdef.behaviour(ind)})\})\}) \text{ in } \)
8. \( \text{let } locenv-s = \{\text{StateReset}(\rho, \text{locals}, N) \mid \}
9. \( \text{locals} \in VS[\text{agdef.localdefs}](\rho.Genenv) \text{ in } \)
10. \( \{(\text{agdef.behaviour(i).body,} \)} \)
11. \( \mu(\rho, Cenv \mapsto \text{AddEnv(locenv, bind-env)}, \}
12. \( \text{Gaenv} \mapsto \text{Gaenv, cae } \mapsto N)\}) \mid \)
13. \( (i, \text{bind-env}) \in \text{ibenv-s} \land \)
14. \( \text{locenv} \in \text{locenv-s} \}

\( \forall \forall (B, \rho') \in SE(\text{mk-Agentservice}(N, \text{expr}), \rho). \)

\[ C \to (\text{mk-AgentService}(N, \text{expr}), \rho) \Rightarrow (B, \rho') \]

Annotations to Rule AS-ve:

There is no hypothesis. The rules transform a sts consisting of an agent service and an environment \( \rho \) into a sts consisting of a behaviour of the agent and a possibly enriched environment.

2 First the current value environment and the agent definition matching the agent service construction are set.

3–8 A set of tuples is constructed. Each tuple consists of an index, fixing a behaviour in the sequence of behaviours associated with the agent \( N \), and an environment containing the bindings that resulted from a match of the associated valpattern against a value that resulted from evaluating \( expr \).

6–8 A set of tuples with type \( N \times LPatEval \) is constructed. Each tuple resembles an agent behaviour construction, denoted by the index \( ind \) and a loose pattern evaluator which is a set of pattern evaluators, that resulted from the evaluation of the behaviour value pattern. A pattern evaluator \( PatEval \) is a total function from values to binding evaluators (see the semantic domains).

6 From the set of tuples the looseness in the second component of the tuples is propagated one level out, resulting in a set of tuples with type \( N \times PatEval \). Now, each tuple represents an agent behaviour construction with a deterministic pattern.
The set of matching values is constructed. The result of $VS[expr]$ is a loose expression evaluator, i.e., a set of expression evaluators. Each expression evaluator has type $LEEval$, a total function from an environment to a value. The set of matching values is then constructed by applying each expression evaluator $eval$ to the current environment $env$.

Again a set of tuples is constructed, each with type $N \times BindEval$-set. The second component collects all binding evaluators, which are total functions from environments to environments. A binding evaluator originates from applying a value to a pattern evaluator.

Again the looseness in the second component from the set of tuples is propagated one level out, resulting in a set of tuples with type $N \times BindEval$.

Each element in this set is transformed into a tuple with type $N \times ENV$, representing a behaviour expression and the set of bindings $\{id \mapsto val\}$ that resulted from the pattern matching process in line 4–8. A $BindEval$ results in a set of environments, so finally we need one more $LP$ to propagate the multiplicity of the second component.

A set of local value-environments is constructed by adding the models for the local definitions, evaluated in the context of the initial value-environment, to the current value-environment $env$, followed by a conditional state reset. If the current agent-environment is equal to the new agent-environment, denoted by $N$, the state values are retained. Otherwise, the state values are re-initialised.

Finally the set of state-transition states $(B, \rho_1)$ is constructed that forms the basis for the agent service transition rule. The behaviour expression component is fixed by the index $i$ in the set of tuples $ibenv$-s. The associated environment is formed out of the current environment $\rho$ in which the value-environment is replaced with a new one constructed from the binding-environment that resulted from pattern matching $bind.env$ and a local value-environment from the set $locenv$-s. The current agent identifier is updated.

Rule C.3  AgentService without value expression
## C.1. Loose environment manipulation rules

<table>
<thead>
<tr>
<th>rule AS-nv</th>
</tr>
</thead>
</table>
| 1 \[ SE(mk-Agentservice(N, \text{nil}), \rho) \triangleq \]
| 2 \hspace{1em} \text{let } env = \rho.Cvenv, agdef = \rho.Gaenv(N) \text{ in} \\
| 3 \hspace{2em} \text{let } locenv-s = \{ \text{StateReset}(\rho, \text{locals}, N) \} \\
| 4 \hspace{3em} \text{locals} \in VS[agdef.localdefs](\rho.Gvenv) \text{ in} \\
| 5 \hspace{4em} \{ ((\text{cases } agdef:} \\
| 6 \hspace{5em} \text{mk-Agdef}(vp, p, b, l) \rightarrow agdef.behaviour(1).body, \\
| 7 \hspace{6em} \text{mk-LAB}(a) \rightarrow agdef.agent), \\
| 8 \hspace{7em} \mu(\rho, Cvenv \mapsto \text{locenv}, \\
| 9 \hspace{8em} Caenv \mapsto Gaenv, cae \mapsto N) \} \mid \text{locenv} \in \text{locenv-s} \}
| \forall \forall (B, \rho') \in SE(mk-Agentservice(N, \text{nil}), \rho). |

\[ H \]
\[ C \quad 10 \quad (mk-AgentService(N, \text{nil}), \rho) \rightarrow (B, \rho') \]

### Annotations to Rule AS-nv:

8 The value expression component is nil. The static semantics should check that len agdef.behaviour = 1. As there is no value part, there is no need for pattern matching.

3-4 A set of local value-environments is constructed by adding the models for the local definitions to the current value-environment, followed by a conditional state reset. If the current agent environment is equal to the new agent environment, denoted by N, the state values are retained. Otherwise, the state values are re-initialised.

5-9 The set of state-transition states is now formed by taking the behaviour expression and combining it with all possible environments from the set locenv-s. If the behaviour is defined by an agent definition the behaviour component is selected, else the behaviour is defined by an local agent binding, and the behaviour is in the agent component.
Rule C.4  UnitService with value expression

<table>
<thead>
<tr>
<th>rules US-ve</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( SE(mk\text{-}UnitService(U, Item, expr), \rho) \triangleq )</td>
</tr>
<tr>
<td>2 let ( env = \rho.Cenv, Unit = \rho.Genv(U) ) in</td>
</tr>
<tr>
<td>3 let locenv-s = { StateReset(\rho, locals, U)</td>
</tr>
<tr>
<td>4 ( locals \in VS<a href="%5Crho.Genv">Unit.localdefs</a> } ) in</td>
</tr>
<tr>
<td>5 let benv-s = { be(env)</td>
</tr>
<tr>
<td>6 ( LP({ pateval(v)</td>
</tr>
<tr>
<td>7 v \in { eeval(env)</td>
</tr>
<tr>
<td>8 pateval \in VS[Unit.share(Item).behaviour.valpattern], }) }</td>
</tr>
<tr>
<td>9 in { (Unit.share(Item).behaviour.body,</td>
</tr>
<tr>
<td>10 \mu(\rho, Cenv \mapsto AddEnv(locenv, bind-env),</td>
</tr>
<tr>
<td>11 Caenv \mapsto Gaenv, cae \mapsto U)</td>
</tr>
<tr>
<td>12 bind-env \in benv-s \land</td>
</tr>
<tr>
<td>13 locenv \in locenv-s }</td>
</tr>
<tr>
<td>14 \forall B, \rho' \in SE(mk\text{-}UnitService(N, Item, expr), \rho) \Rightarrow B, \rho'</td>
</tr>
</tbody>
</table>

Annotations to Rule US-ve:

2 First the current value environment and the unit definition matching the
unit service construction are set.

5-8 Unit items have unique names and their behaviour cannot be multiple
defined as with agent behaviours. So the construction of the set of binding
environments is more straightforward here.

8 The set of pattern evaluators is constructed.

7 The set of matching values is constructed. The result of \( VS[expr] \) is a loose
expression evaluator, i.e., a set of expression evaluators.

6-8 Applying a pattern evaluator to a value results in a binding evaluator. The
set of all possible binding evaluators, given a set of values and a set of
pattern evaluators, is constructed.

5 A set of binding environments is created by the application of the binding
evaluators to the current environment.

3-4 A set of local value environments is constructed by adding the models for
the local unit definitions to the current value-environment \( env \), followed by
a conditional state reset. If the current agent environment is an item
within the unit denoted by \( U \), the state values are retained. Otherwise,
the state values are re-initialised.
Finally the set of state-transition elements \((B, \rho_1)\) is constructed that forms the basis for the unit service transition rule.

**Rule C.5** UnitService without value expression

<table>
<thead>
<tr>
<th>rules US-nv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (SE(\text{mk-UnitService}(U, \text{Item}, \text{nil}), \rho) \triangleq)</td>
</tr>
<tr>
<td>2 let (env = \rho.C\text{venv}, \text{Unit} = \rho.G\text{venv}(U)) in</td>
</tr>
<tr>
<td>3 let (loc\text{env-s} = {\text{StateReset}(\text{AddEnv}(\rho.G\text{venv}, \text{locals}), U)\mid)</td>
</tr>
<tr>
<td>4 (\text{locals} \in \forall S[\text{Unit.localdefs}(\rho.G\text{venv})]) in</td>
</tr>
<tr>
<td>5 ({(\text{Unit.share(\text{Item}).behaviour.body,})</td>
</tr>
<tr>
<td>6 (\mu(\rho, C\text{venv} \leftarrow \text{loc\text{environ}},)</td>
</tr>
<tr>
<td>7 (C\text{venv} \leftarrow G\text{venv}, cae \leftarrow U))\mid \text{loc\text{environ} \in loc\text{environ-s}})</td>
</tr>
<tr>
<td>(\forall (B, \rho') \in SE(\text{mk-UnitService}(N, \text{Item}, \text{nil}), \rho)).</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>C (\vdash \text{mk-UnitService}(N, \text{Item}, \text{nil}), \rho) \implies (B, \rho'))</td>
</tr>
</tbody>
</table>

**Annotations to Rule US-nv:**

2 First the current value environment and the unit definition matching the unit service construction are set.

3-4 A set of local value environments is constructed by adding the models for the local unit definitions to the current value-environment \(env\), followed by a conditional state reset. If the current agent environment is an item within the unit denoted by \(U\), the state values are retained. Otherwise, the state values are re-initialised. item[5-7] The quantification set of sts elements is formed.

8 The value expression component is nil. As there is no value part, there is no need for pattern matching.
### Rule C.6 Agent Value Let Expression

<table>
<thead>
<tr>
<th>rules AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $SE(mk-AgentValueLet(locals, in), \rho) \triangleq$</td>
</tr>
<tr>
<td>2 let $env = \rho.Cenv$ in</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4 then $(mk-Divergent(), \rho)$</td>
</tr>
<tr>
<td>5 else $(in, \mu(\rho, Cenv \mapsto AddEnv(env, ld(env))))$</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>$\forall \forall (B, \rho') \in SE(mk-AgentValueLet(locals, in), \rho), (mk-AgentValueLet(locals, in), \rho) \rightarrow (B, \rho')$</td>
</tr>
</tbody>
</table>

Annotations to Rule AVL:

2 The current value environment is selected.

6 A loose definer is created, i.e. the set of all possible models for the local definitions. Each set element is a definer, i.e. a total function on environments, with signature $ENV \rightarrow (ENV \cup \{err\})$.

3–6 For all non-error elements in the resulting environments, a sts is created consisting of the body of the agent let, in, and the enriched environment.

### Rule C.7 Agent Let Be Expression

<table>
<thead>
<tr>
<th>rules ALB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $SE(mk-AgentLetBe(bind, st, in), \rho) \triangleq$</td>
</tr>
<tr>
<td>2 let $env = \rho.Cenv$ in</td>
</tr>
<tr>
<td>3 let $bev = VS[bind]$, $venv-s = {b(env)</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>$\forall \forall (B, \rho') \in SE(mk-AgentLetBe(bind, st, in), \rho), (mk-AgentLetBe(bind, st, in), \rho) \rightarrow (B, \rho')$</td>
</tr>
</tbody>
</table>

Annotations to Rule ALB:


C.1. Loose environment manipulation rules

2 The current value environment is selected.
3 The bind component is evaluated, the result is a set of bind evaluators.
4–8 A set of value environments is constructed, such that the environments contain all the possible bindings of identifiers to values that resulted from bind and are constrained by the st component.
5 The loose bind evaluator bev has type (ENV → VENV-set)-set, from which the looseness in the second component is propagated out by the function LP, resulting in a set with type (ENV → VENV)-set.

Rule C.8 Agent Agent Let Expression

<table>
<thead>
<tr>
<th>rules AAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [ SE(\text{mk-AgentAgentLet}(\text{agents}, \text{in}), \rho) \triangleq { (\text{in}, \mu(\rho, \text{Caenv} \mapsto \text{Caenv} \cup \text{agents})) } ]</td>
</tr>
<tr>
<td>2 [ \forall (B, \rho') \in SE(\text{mk-AgentAgentLet}(\text{agents}, \text{in}), \rho). ]</td>
</tr>
<tr>
<td>3 [ H ]</td>
</tr>
<tr>
<td>4 [ \text{C} ]</td>
</tr>
<tr>
<td>5 [ \text{(mk-AgentAgentLet}(\text{agents}, \text{in}), \rho) \Rightarrow (B, \rho') ]</td>
</tr>
</tbody>
</table>

Annotations to Rule AAL:

2 The behaviour expressions that form the range of the agents map component must be action guarded (which is not formally stated here). The map itself is added to the current environment.

Rule C.9 Prefix with state manipulation

<table>
<thead>
<tr>
<th>rules Prefix-sman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [ SE(\text{mk-Prefix}(\text{mk-SM}(\text{action}, d), P), \rho) \triangleq ]</td>
</tr>
<tr>
<td>2 [ \text{let svar} \equiv \text{dom ( \rho ).Stenv in} ]</td>
</tr>
<tr>
<td>3 [ \text{let env} = (\text{svar &amp; ( \rho ).Cenv}) \cup \rho \cdot \text{Stenv in} ]</td>
</tr>
<tr>
<td>4 [ \text{let nenuv-s} = { \text{nenv}</td>
</tr>
<tr>
<td>5 [ { (P, \mu(\rho, C\text{env} \mapsto \text{nenv}, ]</td>
</tr>
<tr>
<td>6 [ \text{Stenv} \mapsto \text{svar} \triangleq \text{nenv})</td>
</tr>
<tr>
<td>7 [ \forall (B, \rho') \in SE(\text{mk-Prefix}(\text{mk-SM}(\text{action}), P), \rho) ]</td>
</tr>
<tr>
<td>8 [ H ]</td>
</tr>
<tr>
<td>9 [ \text{C} ]</td>
</tr>
<tr>
<td>10 [ \text{(mk-Prefix}(\text{mk-SM}(\text{action}, d), P), \rho) \Rightarrow (\text{mk-Cdt}(d, B), \rho') ]</td>
</tr>
</tbody>
</table>

Annotations to Rule Prefix-sman:

There is no hypothesis. A state manipulation can not perform idle actions. The pattern \( d \) denotes a duration specification of the state manipulation.
2 The current state environment is selected.

3 The current value environment is selected.

4 A set of environments is created, that resulted from the evaluation of action. The statements in action are restricted to those that do not propagate exits out of their textual environment. Values that result from the evaluation of the statement, like e.g. call statements and return statements, are discarded.

5 The new value-environment replaces the current one.

6 The new state environment replaces the current one.

7 The computation involved with the action component takes \( d \) time, which is realised by embedding the continuation behaviour \( B \) in a count-down timer.

## C.2 Loose action rules

### C.2.1 Conditional action expression

**Rule C.10** Agent If Expression

\[
\begin{array}{|c|c|}
\hline
\text{rules AIf} & \\
\hline
1 & \mathcal{SE}(\text{mk-AgentIf}(be, P, Q), \rho) \Delta \\
2 & \text{let } env = \rho. Cenv \text{ in} \\
3 & \text{let } testval-s = \forall S[be](env) \text{ in} \\
4 & \text{let } res-s = \{(\text{cases } tv:\}
5 & \quad \text{True()} \rightarrow (P, \rho) \\
6 & \quad \text{False()} \rightarrow (Q, \rho) \} \mid tv \in testval-s \text{ in} \\
7 & \quad \text{if card } res-s = 0 \\
8 & \quad \text{then } \{(\text{mk-Divergent()}, \rho)\} \\
9 & \quad \text{else } res-s \\
\hline
\forall (B, \rho_B) \in \mathcal{SE}(\text{mk-AgentIf}(be, P, Q), \rho). \\
\hline
\text{H} & 10 \quad (B, \rho_B) \xrightarrow{\alpha} (B', \rho_B') \\
\text{C} & 11 \quad (\text{mk-AgentIf}(be, P, Q), \rho) \xrightarrow{\alpha} (B', \rho_B') \\
\hline
\end{array}
\]

Annotations to Rule AIf:

2 The current value environment is selected.

3 The set of values resulting from the evaluation of the be component is created. Again we have a loose result, which can either be a truth-value or some other value.

4–6 For each truth-value in the set testval-s a sts is added to the result set res-s.
8 If the result set res-s is empty, which can only be the case if the values in testval-s are not boolean values, the resulting behaviour expression in the singleton set will denote divergence.

### C.2.2 Prefix expressions

**Rule C.11** Prefix with Timed InAct - Action

<table>
<thead>
<tr>
<th>Rule C.11</th>
<th>Prefix with Timed InAct - Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rules Prefix-in-action</strong></td>
<td><strong>rules Prefix-in-action</strong></td>
</tr>
<tr>
<td>1 ( \mathcal{P} \mathcal{E}_{\text{in}}((\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho), \text{val}) \triangleq )</td>
<td>1 ( \mathcal{P} \mathcal{E}_{\text{in}}((\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho), \text{val}) \triangleq )</td>
</tr>
<tr>
<td>2 let env = ( \rho.\text{Cenv} ) in</td>
<td>2 let env = ( \rho.\text{Cenv} ) in</td>
</tr>
<tr>
<td>3 let menu-s = union { { be(env)</td>
<td>be \in LP({pev(val)}) } }</td>
</tr>
<tr>
<td>( \uparrow { \text{pev} \in \mathcal{V}S[\text{pat}] } ) in</td>
<td>( \uparrow { \text{pev} \in \mathcal{V}S[\text{pat}] } ) in</td>
</tr>
<tr>
<td>{ P, \mu(\rho, \text{Cenv} \leftarrow \text{AddEnv(env, menu)}) }</td>
<td>{ P, \mu(\rho, \text{Cenv} \leftarrow \text{AddEnv(env, menu)}) }</td>
</tr>
<tr>
<td>( { \text{menu} \in \text{menu-s} } )</td>
<td>( { \text{menu} \in \text{menu-s} } )</td>
</tr>
<tr>
<td>( \forall \forall (B, \rho') \in \mathcal{P} \mathcal{E}_{\text{in}}((\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho), \text{val}) )</td>
<td>( \forall \forall (B, \rho') \in \mathcal{P} \mathcal{E}_{\text{in}}((\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho), \text{val}) )</td>
</tr>
<tr>
<td>( \text{H} )</td>
<td>( \text{H} )</td>
</tr>
<tr>
<td>( \text{C} )</td>
<td>( \text{C} )</td>
</tr>
<tr>
<td>7 ( (\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho) \xrightarrow{\text{lab}#\text{val}} (\text{mk-Cdt}(\mathcal{V}S[\text{d}][\rho.\text{Cenv}], B), \rho') )</td>
<td>7 ( (\text{mk-Prefix}(\text{mk-InAct}(\text{lab}, \text{pat}, d, -), P), \rho) \xrightarrow{\text{lab}#\text{val}} (\text{mk-Cdt}(\mathcal{V}S[\text{d}][\rho.\text{Cenv}], B), \rho') )</td>
</tr>
</tbody>
</table>

**Annotations to Rule Prefix-in-action**:

There is no hypothesis.

2 The current value environment is selected.

3–4 A set of environments is created, that resulted from the matching of the input value \( \text{val} \) against the pattern \( \text{pat} \). The result from applying \( \text{env} \) to the bind evaluator \( \text{be} \) results in a set possible bindings.

3 All sets are collected into one set.

8 The duration specification \( d \) results in a count-down timer that encapsulates the stcs that follows from taking the in action.

**Rule C.12** Prefix with Timed OutAct

<table>
<thead>
<tr>
<th>Rule C.12</th>
<th>Prefix with Timed OutAct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rule Prefix-out-action</strong></td>
<td><strong>rule Prefix-out-action</strong></td>
</tr>
<tr>
<td>1 ( \mathcal{P} \mathcal{E}_{\text{out}}(\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) \triangleq )</td>
<td>1 ( \mathcal{P} \mathcal{E}_{\text{out}}(\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) \triangleq )</td>
</tr>
<tr>
<td>{ \text{eeval}(\rho.\text{Cenv})</td>
<td>\text{eeval} \in \mathcal{V}S[\text{expr}] } )</td>
</tr>
<tr>
<td>( \forall \forall \text{val} \in \mathcal{P} \mathcal{E}_{\text{out}}(\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) )</td>
<td>( \forall \forall \text{val} \in \mathcal{P} \mathcal{E}_{\text{out}}(\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) )</td>
</tr>
<tr>
<td>( \text{H} )</td>
<td>( \text{H} )</td>
</tr>
<tr>
<td>( \text{C} )</td>
<td>( \text{C} )</td>
</tr>
<tr>
<td>3 ( (\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) \xrightarrow{\text{lab}#\text{val}} (\text{mk-Cdt}(d, P), \rho) )</td>
<td>3 ( (\text{mk-Prefix}(\text{mk-Outact}(\text{lab}, \text{expr}, d, -), P), \rho) \xrightarrow{\text{lab}#\text{val}} (\text{mk-Cdt}(d, P), \rho) )</td>
</tr>
</tbody>
</table>
Annotations to **Rule Prefix-out-action** :

1. There is no hypothesis.
2. A set of values is created, that resulted from the evaluation of `expr` in the current value environment.
3. The duration specification `d` results in a count-down timer that encapsulates the `sts` that follows from taking the action.

### C.3 Standard action rules

#### C.3.1 Prefix expression

**Rule C.13** *Prefix with Timed SynAct — Action*

<table>
<thead>
<tr>
<th>rule Prefix-syn-action</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Annotations to **Rule Prefix-syn-act** :

1. There is no hypothesis. An prefix signifies a transition that is always possible.

#### C.3.2 Choice expressions

**Rule C.14** *Left Choice*

<table>
<thead>
<tr>
<th>rule Choice-l</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1</td>
</tr>
<tr>
<td>C 2</td>
</tr>
</tbody>
</table>

**Rule C.15** *Left Choice +*

<table>
<thead>
<tr>
<th>rule Choice-l+</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1</td>
</tr>
<tr>
<td>C 2</td>
</tr>
</tbody>
</table>
C.3. Standard action rules

Annotations to Rule Choice-l:

1–2 Whenever the left operand of the binary choice operator has an $x$ transition on the environment of the choice, the whole choice has the same $x$ transition. The choice is a binary form of the more general summation construction in Milner’s CCS [157].

Rule C.16 Right Choice

<table>
<thead>
<tr>
<th>Rule Choice-r</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 $(Q, \rho) \xrightarrow{x} (Q', \rho_q)$</td>
</tr>
<tr>
<td>C 2 $(mk\text{-}Choice(P, Q), \rho) \xrightarrow{x} (Q', \rho_q)$</td>
</tr>
</tbody>
</table>

Rule C.17 Right Choice +

<table>
<thead>
<tr>
<th>Rule Choice-r+</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 $(Q, \rho_q) \xrightarrow{x} (Q', \rho'_q)$</td>
</tr>
<tr>
<td>C 2 $(mk\text{-}Choice(P, Q), \rho_p + \rho_q) \xrightarrow{x} (Q', \rho'_q)$</td>
</tr>
</tbody>
</table>

Annotations to Rule Choice-r:

1–2 Whenever the right operand of the binary choice operator has an $x$ transition on the environment of the choice, the whole choice has the same $x$ transition.

C.3.3 Composition expressions

Rule C.18 Left Composition Intro

<table>
<thead>
<tr>
<th>Rule Composition-l intro</th>
</tr>
</thead>
<tbody>
<tr>
<td>H 1 $(P, \rho) \xrightarrow{x} (P', \rho_p)$</td>
</tr>
<tr>
<td>C 2 $(mk\text{-}Composition(P, Q), \rho) \xrightarrow{x}$</td>
</tr>
<tr>
<td>3 $(mk\text{-}Composition(P', Q), \rho_p</td>
</tr>
</tbody>
</table>

Annotations to Rule Composition-l:

1 Whenever the left operand of the binary composition operator has an $x$ transition on the environment of the composition,

2–3 the whole composition has the same $x$ transition. The possible side-effects due to a state-manipulation action are registered in the original environment $\rho$. 
Rule C.19  \textit{Left Composition}

<table>
<thead>
<tr>
<th>rule Composition-l</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Annotations to Rule Composition-l:

1. Whenever the left operand of the binary composition operator has an \( x \) transition on the environment of the composition,
2. and 3. the whole composition has the same \( x \) transition. The possible side-effects due to a state-manipulation action are registered in the original environment \( \rho_q \).

Rule C.20  \textit{Right Composition Intro}

<table>
<thead>
<tr>
<th>rule Composition-r intro</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Annotations to Rule Composition-r:

1. Whenever the right operand of the binary composition operator has an \( x \) transition on the environment of the composition,
2. and 3. the whole composition has the same \( x \) transition. The possible side-effects due to a state-manipulation action are registered in the original environment \( \rho \).

Rule C.21  \textit{Right Composition}

<table>
<thead>
<tr>
<th>rule Composition-r</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Annotations to Rule Composition-r:

1. Whenever the right operand of the binary composition operator has an \( x \) transition on the environment of the composition,
2–3 the whole composition has the same \(x\) transition. The possible side-effects due to a state-manipulation action are registered in the original environment \(\rho_p\).

**Rule C.22 Composition with synchronization intro**

<table>
<thead>
<tr>
<th>rule Composition-s intro</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Annotations to Rule Composition-s intro:**

1. Whenever the left operand of the composition has an action on the environment of the composition and the right operand has an co-labeled action on the same environment,

2. the whole composition has an internal action resulting in a environment containing the two resulting environments.

**Rule C.23 Composition with synchronization**

<table>
<thead>
<tr>
<th>rule Composition-s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Annotations to Rule Composition-s**:

1. Whenever the left operand of the composition has an action on its environment and the right operand has an co-labeled action on its environment,

2. the whole composition has an internal action resulting in an composite environment containing the two resulting environments.

**C.3.4 Restriction and relabelling expressions**

**Rule C.24 Restriction**

<table>
<thead>
<tr>
<th>rule Restr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong></td>
</tr>
<tr>
<td><strong>C</strong></td>
</tr>
</tbody>
</table>
Annotations to Rule Restr:

1. Whenever the left operand of the binary restriction operator has an $x$ transition on the environment of the composition, and the label $x$ is not in the restriction set, then
2. the whole restriction has the same $x$ transition.

**Rule C.25 Relabelling**

<table>
<thead>
<tr>
<th>Rule Relab</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

Annotations to Rule Relab:

1. Whenever the left operand of the binary relabelling operator has an $x$ transition on the environment of the composition, and the label $x$ is in the relabel map set, then
2. the whole relabelling has a $y$ transition, where $y = rmap(x)$.

### C.4 Idling rules

**Rule C.26 Prefix with Timed SynAct - Idling**

<table>
<thead>
<tr>
<th>Rule Prefix-syn-idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $TDLE(tid, \rho)\triangle$</td>
</tr>
<tr>
<td>2 let $t = \text{if } tid \in \text{dom } \rho.Cenv$</td>
</tr>
<tr>
<td>3 then $\rho.Cenv(tid)$</td>
</tr>
<tr>
<td>4 else 0 in</td>
</tr>
<tr>
<td>5 $\mu(\rho, Cenv \mapsto \rho.Cenv \cup {tid \mapsto t + d})$</td>
</tr>
</tbody>
</table>

| H | $6 (mk\text{-Prefix}(mk\text{-SynAct}(lab, tid), P), \rho) \xrightarrow{(d)}$ |
| C | $7 (mk\text{-Prefix}(mk\text{-SynAct}(lab, tid), P), TDLE(tid, \rho))$ |

Annotations to Rule Prefix-syn-idle:

2–5 The idle time $d$ is added to the registration of the cumulative idling of the prefix.

**Rule C.27 Prefix with Timed InAct - Idling**
### Rule Prefix-in-idle

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( TDL(t, \rho) \triangleq )</td>
</tr>
<tr>
<td>2</td>
<td>let ( t = ) if ( tid \in \text{dom} \rho.Cenv )</td>
</tr>
<tr>
<td>3</td>
<td>then ( \rho.Cenv(tid) )</td>
</tr>
<tr>
<td>4</td>
<td>else 0 in</td>
</tr>
<tr>
<td>5</td>
<td>( \mu(\rho, Cenv \mapsto \rho.Cenv \cup {tid \mapsto t + d}) )</td>
</tr>
</tbody>
</table>

### Annotations to Rule Prefix-in-idle:
2–5 The idle time \( d \) is added to the registration of the cumulative idling of the prefix.

### Rule C.28 Prefix with Timed OutAct - Idling

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( TDL(t, \rho) \triangleq )</td>
</tr>
<tr>
<td>2</td>
<td>let ( t = ) if ( tid \in \text{dom} \rho.Cenv )</td>
</tr>
<tr>
<td>3</td>
<td>then ( \rho.Cenv(tid) )</td>
</tr>
<tr>
<td>4</td>
<td>else 0 in</td>
</tr>
<tr>
<td>5</td>
<td>( \mu(\rho, Cenv \mapsto \rho.Cenv \cup {tid \mapsto t + d}) )</td>
</tr>
</tbody>
</table>

### Annotations to Rule Prefix-out-idle:
2–5 The idle time \( d \) is added to the registration of the cumulative idling of the prefix.

### Rule C.29 Prefix with IdleAct — Init

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \text{DELAY}(\text{mk-Prefix}(\text{mk-IdleAct}(delay), P), \rho) \triangleq )</td>
</tr>
<tr>
<td>2</td>
<td>( \text{VS}<a href="%5Crho.Cenv">delay</a> )</td>
</tr>
<tr>
<td>3</td>
<td>( \forall dv \in \text{DELAY}(\text{mk-Prefix}(\text{mk-IdleAct}(delay), P), \rho) )</td>
</tr>
</tbody>
</table>

### Annotations to Rule Prefix-idle-init:
3 \( \text{mk-Cdt}(dv, P), \rho) \)
Annotations to Rule Prefix-idle-init:

There is no hypothesis.

2 The delay expression is evaluated and

3 for each value in the resulting sets of denotations an internal action creates a count-down timer.

Rule C.30 Prefix with IdleAct — Idling

\[
\begin{array}{|c|c|}
\hline
\text{rules Prefix-idle-idling} & \\
\hline
1 & CDT(mk-Prefix(mk-Cdt(dv, P), \rho)) \triangleleft dv \\
2 & \\
\hline
\forall \forall d < CDT(mk-Prefix(mk-Cdt(dv, P), \rho)) & \\
\hline
H & \\
\hline
C & (mk-Cdt(dv, P), \rho) \xrightarrow{\epsilon(d)} (mk-Cdt(dv - d, P), \rho) \\
\hline
\end{array}
\]

Annotations to Rule Prefix-idle-idling:

There is no hypothesis. This is a rule template. For each value \( d < dv \) there is a possible transition for the count-down counter.

Rule C.31 Prefix with IdleAct — End

\[
\begin{array}{|c|c|}
\hline
\text{rule Prefix-idle-end} & \\
\hline
H & \\
\hline
C & (mk-Cdt(dv, P), \rho) \xrightarrow{\epsilon(dv)} (P, \rho') \\
\hline
\end{array}
\]

Annotations to Rule Prefix-idle-end:

1 When the count-down counter reaches 0, the idle action of the prefix is completed and agent \( P \) continues.

Rule C.32 Prefix with IdleAct — Idle prefix

\[
\begin{array}{|c|c|}
\hline
\text{rule Prefix-idle-idle-prefix} & \\
\hline
H & (P, \rho) \xrightarrow{\epsilon(d)} (P', \rho') \\
\hline
C & (mk-Cdt(dv, P), \rho) \xrightarrow{\epsilon(d + dv)} (P', \rho') \\
\hline
\end{array}
\]
Annotations to **Rule Prefix-idle-idle-prefix**:

1. If the behaviour expression $P$ may idle to $P'$,
2. then the count down counter prefix to $P$ transforms by idling into $P'$ in one cumulative step.

**Rule C.33 Idling Choice Intro**

<table>
<thead>
<tr>
<th>rule Choice-idle(+intro)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong> 1 $(P, \rho) \xrightarrow{\epsilon^{(d)}} (P', \rho_p) \land (Q, \rho) \xrightarrow{\epsilon^{(d)}} (Q', \rho_q)$</td>
</tr>
<tr>
<td><strong>C</strong> 2 $(mk\text{-Choice}(P, Q), \rho) \xrightarrow{\epsilon^{(d)}} (P', Q', \rho_p + \rho_q)$</td>
</tr>
</tbody>
</table>

**Rule C.34 Idling Choice**

<table>
<thead>
<tr>
<th>rule Choice-idle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong> 1 $(P, \rho_p) \xrightarrow{\epsilon^{(d)}} (P', \rho'_p) \land (Q, \rho_q) \xrightarrow{\epsilon^{(d)}} (Q', \rho'_q)$</td>
</tr>
<tr>
<td><strong>C</strong> 2 $(mk\text{-Choice}(P, Q), \rho_p + \rho_q) \xrightarrow{\epsilon^{(d)}} (P', Q', \rho'_p + \rho'_q)$</td>
</tr>
</tbody>
</table>

Annotations to **Rule Choice-idle**:

1. Whenever the left operand and the right operand of the binary choice operator idle,
2. the whole choice idles.

**Rule C.35 Idling Composition Intro**

<table>
<thead>
<tr>
<th>rule Composition-idle(− intro)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong> 1 $(P, \rho) \xrightarrow{\epsilon^{(d)}} (P', \rho_p) \land (Q, \rho) \xrightarrow{\epsilon^{(d)}} (Q', \rho_q) \land Sort_d(P) \cap Sort_d(Q) = 0$</td>
</tr>
<tr>
<td>2 $(mk\text{-Composition}(P, Q), \rho_p</td>
</tr>
<tr>
<td><strong>C</strong> 3 $(mk\text{-Composition}(P', Q'), \rho_p</td>
</tr>
</tbody>
</table>

**Rule C.36 Idling Composition**

<table>
<thead>
<tr>
<th>rule Composition-idle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H</strong> 1 $(P, \rho_p) \xrightarrow{\epsilon^{(d)}} (P', \rho'_p) \land (Q, \rho_q) \xrightarrow{\epsilon^{(d)}} (Q', \rho'_q) \land Sort_d(P) \cap Sort_d(Q) = 0$</td>
</tr>
<tr>
<td><strong>C</strong> 3 $(mk\text{-Composition}(P, Q), \rho_p</td>
</tr>
<tr>
<td>4 $(mk\text{-Composition}(P', Q'), \rho'_p</td>
</tr>
</tbody>
</table>
Annotations to Rule Composition idling:

1 Whenever the left operand and the right operand can idle equal amounts of time, and
2 within that amount of time the operands cannot come to a synchronization, then
3 the composition may idle that amount of time.

C.5 Null agents and Divergence

Rule C.37 NullAgent

<table>
<thead>
<tr>
<th>rules Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall d \in T )</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>C 1</td>
</tr>
<tr>
<td>( (\text{mk-NullAgent}(val), \rho) \xrightarrow{\in(d)} (\text{mk-NullAgent}(val), \rho) )</td>
</tr>
</tbody>
</table>

Annotations to Rule Null:

1 An sts in which the behaviour expression consists of the null agent has only time-progressing transitions

Rule C.38 Divergent Agent

<table>
<thead>
<tr>
<th>rule Divergent Internal</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
</tr>
<tr>
<td>C 1</td>
</tr>
<tr>
<td>( (\text{mk-Divergent}(), \rho) \xrightarrow{\iota} (\text{mk-Divergent}(), \rho) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rules Divergent Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall d \in T )</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>C 2</td>
</tr>
<tr>
<td>( (\text{mk-Divergent}(), \rho) \xrightarrow{\in(d)} (\text{mk-Divergent}(), \rho) )</td>
</tr>
</tbody>
</table>

Annotations to Rule Diver:

1 The divergent process is continuously busy with internal \( \iota \) actions or
2 it may take time progressing actions.
Bibliography


Summary

The Vienna Development Method (VDM for short) is a formal method for the specification and subsequent development of sequential, single threaded, state based software components. It is a model-oriented method, i.e., a VDM specification consists of an explicit model of the software component being constructed. VDM consists of two major components: a language and a method. The VDM Specification Language (or VDM-SL for short) supplies a notation in which VDM specifications can be expressed. The VDM method comprises a set of guidelines and techniques for the development starting from a specification into an implementation in a certain programming language.

The VDM notation offers both data abstraction and control abstraction. The data abstraction is concerned with the description and manipulation of data values. The control abstraction offers constructions on micro and macro level, from simple conditional expressions and statements to functional and procedural abstractions. However, all constructions are limited to manipulate the control of a single thread. There are no process abstractions available in the notation.

The algebraic approach to process specification captured in CCS offers a means to describe the simultaneous behaviour of multiple control threads by virtue of behaviour expressions build from actions and operations on actions, like the prefix operator ‘O’.

A behaviour expression is not equivalent to a expression concerning data manipulation with respect to its denotation. Data manipulating expressions denote numerical values, which e.g., result from a computation, by evaluating the expression according to a fixed set of rules. E.g., the expression ‘4 + 5’ computes the value ‘9’ by evaluating it according to rules for the ‘+’ operator. The meaning of data manipulating expressions is fixed by a denotational semantics.
Behaviour expressions like \( p \odot X \) cannot be said to denote a numerical value, that can be \emph{computed} according to rules for the \( \odot \) operator.

One particular semantic approach takes the view that a behaviour expression denotes a state from which possible \emph{transitions}, dictated by a set of transition rules, are possible. These transitions are part of a transition system. This approach builds on the theory of Structured Operational Semantics (SOS).

Traditionally, real-time systems and their software have been considered to be closely connected to time and its properties. Quite often the presence of explicit time in a computer system has been considered the main property of real-time systems. The notion of time seems to be a suitable tool to connect the software components of real-time systems with the rest of the components in the systems.

With the strong expansion in numbers of distributed and real-time applications and the still growing demands on verified properties of these systems, formal theories dedicated to the description of time-dependent behaviour have become highly popular.

This thesis proposes a formal specification language, \textsc{mosca} based on VDM and CCS extended with timing constructions. It offers a SOS semantics for the notation in which the denotational semantics of VDM-SL is embedded. The SOS rules handle the (loose) process and timing constructions. The embedded VDM-SL semantics handles all value manipulation.

\textsc{mosca} is pictured within the software engineering life cycle by investigation of specification, verification and implementation aspects of the language. The thesis concentrates on the notational aspects of the combination without addressing the methodological aspects, which is considered a research topic on its own.
Samenvatting

De Vienna Development Method (VDM) is een formele methode voor de specificatie en ontwikkeling van sequentiële programmatuurcomponenten gebaseerd op een enkelvoudige controldraad. Het is een model-georiënteerde methode. Een VDM specificatie omschrijft een model van de te construeren softwarecomponent. VDM omvat zowel een notatie als een methode. VDM-SL is de specificatienotatie waarmee de softwarecomponenten worden beschreven. De VDM methode omvat een verzameling van richtlijnen en technieken voor de ontwikkeling van een implementatie van de specificaties.

De VDM notatie ondersteunt zowel data- als controlabstractie. De data abstractie omvat zowel de beschrijving als de manipulatie van datawaarden. De controlabstractie is mogelijk op micro- als macroniveau en omvat zowel conditionele expressies en statements als functionele en procedurele abstracties. Echter alle controlconstructies zijn beperkt tot het manipuleren van één controldraad. Er zijn geen mogelijkheden voor het beschrijven van meerdere processen.

De algebraïsche benadering voor procesbeschrijving zoals bevat in de CCS notatie maakt het mogelijk een verzameling van processen te beschrijven door gebruik te maken van gedragsexpressies opgebouwd uit akties en operaties op akties zoals de prefix operator '○'.

Een gedragsexpressie is niet equivalent met een expressie welke data manipulatie beschrijft. Deze laatste soort expressies duiden numerieke waarden welke tot stand komen door een berekening volgens een vast stel regels. Zo levert de evaluatie van de expressie '4+5' de numerieke waarde '9' door berekening volgens de regels van de '+' operator. De betekenis van de data-expressies is vastgelegd door middel van een denotationele semantiek.

Gedragsexpressies zoals 'p ⊕ X' beschrijven geen numerieke waarde die berekend kan worden door de regels van de '○' operator. Een bepaalde semantische benadering beschouwt gedragsexpressies als een toestand van
waaruit transities mogelijk zijn naar andere toestanden, bepaald door een verzameling van transitieregels. Deze regels zijn deel van een transitiesysteem. Deze benadering bouwt op de theorie van Structurele Operationele Semantiek (SOS).

Real-time systemen en real-time software zijn sterk verbonden met het begrip tijd. De realisatie van een expliciet tijdsbegrip in een computersysteem is vaak beschouwd als het meest specifieke aspect van real-time systemen. Het tijdsbegrip vormt zo een geschikt gereedschap als bindende factor tussen de programmatuur componenten en de overige componenten van een real-time systeem.

Door de sterke toename van gedistribueerde- en real-time applicaties en de steeds grotere behoefte aan geverifieerde eigenschappen van deze systemen zijn formele technieken gewijd aan de beschrijving van tijdsafhankelijk gedrag sterk in de belangstelling.

Dit promotiewerk presenteert een formele specificatietaal, MOSCA gebaseerd op VDM en CCS en uitgebreid met tijdsbeschrijvende constructies. Het biedt een SOS semantiek voor de notatie waarin de denotationele semantiek van VDM-SL is ingebed. De SOS regels beschrijven de (mogelijk los gespecificeerde) processconstructies en de tijdsconstructies. De ingebedde VDM-SL semantiek beschrijft de datamanipulatieconstructies.

De rol van MOSCA binnen de levenscyclus van programmatuur wordt onderzocht door inspectie van de specificatie-, verificatie- en implementatie-eigenschappen van de notatie. De thesis concentreert zich op de studie van de notationele eigenschappen van de specificatietaal zonder bestudering van de methodologische aspecten van de combinatie. Dit laatste wordt beschouwd als een zelfstandig promotieonderzoek.
Curriculum vitae

Hans Toetenel studied mathematics and computer science and graduated from the Delft University of Technology in 1974. After a short period at a division of Philips Telecommunication Industry where he worked as a software engineer he returned to Delft to study and teach principles of programming languages and their implementation at the Faculty of Technical Mathematics and Informatics. Earlier work in the field of real-time computing was directed at the study and implementation of run-time support facilities for the Ada tasking model. His recent work has been dedicated to formal specification notations and their rôle within software production. He has published in the fields of programming languages, formal specification languages and software engineering.